A Structured Approach to Parallel Programming

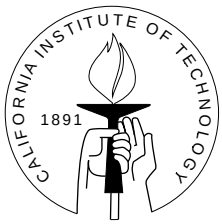
Thesis by

Berna Massingill

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy



California Institute of Technology

Pasadena, California

1998

(Submitted September 25, 1997)

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
1998		**00-00-1998 to 00-00-1998**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
A Structured Approach to Parallel Programming	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Air Force Office of Scientific Research,875 North Randolph Street Suite 325,Arlington,VA,22203-1768**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**184**	

Acknowledgments

What a long, strange trip it's been.
— Jerry Garcia

You certainly have met some interesting people out there!
— Billie S. Massingill

Many people contributed, in many different ways, to this thesis and to my education at Caltech. I thank all of them, in particular:

My advisor, Mani Chandy, whose remarkable ability to identify fundamental concepts made my years at Caltech enlightening, and whose patience and all-around niceness made them enjoyable. It has been a pleasure and a privilege to work with him.

The other members of my committee — Jim Arvo, Alain Martin, Dan Meiron, and Eric Van de Velde — who reviewed my thesis and whose questions and comments on both the thesis and the work were insightful and illuminating.

The members of my candidacy committee — Mani, Alain, Dan, Eric, and Yaser Abu-Mostafa — who provided critical feedback at that milestone.

My instructors at Caltech, each of whom contributed a fresh and valuable perspective on his or her field, most memorably Yaser Abu-Mostafa, Alain Martin, Beverly Sanders, Chuck Seitz, Eric Van de Velde, Rick Wilson, and of course Mani. Eric deserves special thanks for his book *Concurrent Scientific Computing*, which was a notable inspiration for this work.

Current and former members of the CS support staff — Dian De Sha, Arlene DesJardins, Cindy Ferrini, Diane Goodfellow, Yvonne Recendez, Patty Renstrom, Gail Stowers, and Nancy Zachariasen — who smoothed my path in countless ways, and all of the other members of the Caltech community — from computer system administrators to the folks in the graduate office — who together created a supportive and friendly environment.

The members of Mani's research group — Joe Kiniry, Adam Rifkin, Eve Schooler, Paul Sivilotti, John Thornley, and Dan Zimmerman — each of whom provided his or her own unique variety of support, from technical expertise and critical feedback on my work to moral support. Adam and Paul deserve special thanks for helping me set up my thesis defense from afar.

The many others past and present members of the department who contributed to the whole experience, among them: Nan Boden, who guided me through the admissions process and in doing so answered more questions than she probably thought one person could ask; Peter Hofstee and Rajit Manohar, who were always ready with both intellectual stimulation and moral support; Wen-King Su, whose technical expertise even my endless questions haven't exhausted and whose sympathetic ear has been equally valuable; Marcel van der Goot, whose TEX expertise I also couldn't exhaust; and the late Mike Pertel, whose behavior in tragic circumstances was an inspiration.

The people in my former life who motivated me to attend Caltech and made it possible for me to do so — Bill Athas, Bob Boyer, and Mike McCants.

The employees of Phillips Laboratory, the Air Force lab where I spent an instructive six weeks one summer. Leon Chandler, who supervised my visit, and John Beggs, who helped with my research project, deserve special thanks, as does in a different way the unnamed physicist whose casual remark about computer scientists — "I can do physics and I can write Fortran; what do I need you guys for?" — prodded me to consider the question of who benefits from my work.

The people who provided material for or participated in the development of applications discussed in this thesis, including John Beggs, Tzu-Yi Chen, Donald Dabdub, Greg Davis, Karl Kunz and Raymond Luebbers, John Langford and Lena Petrovic, Rajit Manohar, Anita Mareno, Dan Meiron, and Ravi Samtaney. Dan deserves special thanks as the joint leader (with Mani) of the long-term collaboration on the role of archetypes in scientific computing that produced most of these applications.

Last but not least, the friends and family without whose support this whole venture would have been difficult if not impossible. My mother deserves special mention for the many, many hours she has spent lending a sympathetic ear and generally doing her best to keep me relatively sane. I might have been able to do it without them, but I'm glad I didn't have to.

The research described in this thesis was funded in part by a Milton E. Mohr graduate fellowship, in part by an Air Force Laboratory graduate fellowship, and in part by the AFOSR and the NSF. I thank them all for their support.

Abstract

Parallel programs are more difficult to develop and reason about than sequential programs. There are two broad classes of parallel programs: (1) programs whose specifications describe ongoing behavior and interaction with an environment, and (2) programs whose specifications describe the relation between initial and final states. This thesis presents a simple, structured approach to developing parallel programs of the latter class that allows much of the work of development and reasoning to be done using the same techniques and tools used for sequential programs. In this approach, programs are initially developed in a primary programming model that combines the standard sequential model with a restricted form of parallel composition that is semantically equivalent to sequential composition. Such programs can be reasoned about using sequential techniques and executed sequentially for testing. They are then transformed for execution on typical parallel architectures via a sequence of semantics-preserving transformations, making use of two secondary programming models, both based on parallel composition with barrier synchronization and one incorporating data partitioning. The transformation process for a particular program is typically guided and assisted by a *parallel programming archetype*, an abstraction that captures the commonality of a class of programs with similar computational features and provides a class-specific strategy for producing efficient parallel programs. Transformations may be applied manually or via a parallelizing compiler. Correctness of transformations within the primary programming model is proved using standard sequential techniques. Correctness of transformations between the programming models and between the models and practical programming languages is proved using a state-transition-based operational model.

This thesis presents: (1) the primary and secondary programming models, (2) an operational model that provides a common framework for reasoning about programs in all three models, (3) a collection of example program transformations with arguments for their correctness, and (4) two groups of experiments in which our overall approach was used to develop example applications. The specific contribution of this work is to present a unified theory/practice framework for this approach to parallel program development, tying together the underlying theory, the program transformations, and the program-development methodology.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is almost an article of faith in the parallel-programming community that parallel programming is significantly more difficult than sequential programming, and that anything one can do to reduce the difficulty of parallel programming is therefore a good thing. There is less agreement about how this can best be done; approaches range from new programming languages to compilers that automatically parallelize sequential programs. The difficulties are perhaps most severe for programs whose specifications are in terms of ongoing behavior and interaction with an environment, since such programs more obviously require tools and techniques other than or in addition to those used for sequential programs. But even programs whose specifications are in terms of the relation between initial and final states — that is, programs that are implemented in parallel primarily for reasons of performance — present difficulties in addition to those encountered in developing their sequential counterparts. This thesis presents a structured approach to the latter class of parallel programs that allows much of the work of development, reasoning, and testing and debugging to be done using familiar sequential techniques and tools. This approach takes the form of a simple model of parallel programming, a methodology for transforming programs in this model into programs for parallel machines based on the ideas of semantics-preserving transformations and programming archetypes (patterns), and an underlying operational model providing a unified framework for reasoning about the requisite transformations. The specific contribution of the thesis is the integration of the operational model, the programming models, and the methodology, all of which build on and exploit existing work, into a unified theory/practice framework for developing and reasoning about parallel programs.

1.1 A little history

This work has its origins in two experimental projects, one exploring the use of archetypes or patterns in developing parallel scientific applications, and one exploring the use of semantics-preserving transformations in parallelizing sequential code.

1.1.1 Experiments with archetypes (patterns)

Our investigation of the use of patterns in developing parallel scientific applications began as a search for what we called *archetypes for parallel scientific computing*. By *archetype*, we mean an abstraction that captures the common features of a class of problems with common computational structure. This idea is useful when applied to traditional computer science algorithms (sorting, searching, graph algorithms, and so forth), so we proposed to experiment with applying it to parallel scientific computing. What we found was that the useful commonality tended to focus on patterns of communication in the parallel versions of applications, so we focused attention on a few representative classes of problems and developed *archetype implementations*, each combining tutorial documentation with a code library encapsulating the communication operations (the "hard parts" of developing a parallel version of an application). We then used these archetype implementations in developing example applications and found that they did assist in the development process.

1.1.2 Experiments with stepwise parallelization

Our investigation of the use of semantics-preserving transformations in parallelizing sequential code consisted of developing a methodology by which a sequential application program could be transformed into an equivalent parallel program via a sequence of small transformations, with all but the last transformation performed in the sequential domain and the final transformation into the parallel domain justifiable via a formal proof applicable to all programs meeting certain stated criteria. With this methodology, all but the final transformation could be checked by testing and debugging in the sequential domain, and since the final transformation had been formally proved to preserve correctness, there would be no need to debug the parallel program. We applied this methodology to two application programs and found that indeed debugging was confined to the sequential versions of the program, with the formally-proved final transformation preserving correctness.

1.1.3 Theoretical framework

We then turned our attention to developing a theoretical framework that would encompass and support both these investigations. The goal of this theoretical work was something simple and applicable to widely-used practical languages, and yet mathematically rigorous, that could serve as a

theoretical support for the experimental work. Our approach was to develop a model and methodology for parallel programming that to a large extent make it possible to develop and reason about parallel programs using the same methods and tools used to develop and reason about sequential programs, together with an operational model that allows us to reason formally about aspects that are not amenable to sequential techniques.

1.2 Related work

1.2.1 Foundations

Sequential programming models and specifications. We define our programming models as simple extensions to the standard sequential model of Dijkstra [36, 37], Gries [42], and others. We base our notions of program correctness on the work of Hoare [44] and others.

Program development via stepwise refinement. Our approach to program development is based on stepwise refinement and program transformations, as described for sequential programs in the work of Back [6], Gries, and Hoare [44], and for parallel programs in the work of, for example, Back [5], Martin [56], and Van de Velde [74].

Operational models. Our operational model is based on defining programs as state-transition systems, as in the work of Chandy and Misra [24], Lynch and Tuttle [52], Lamport [51], Manna and Pnueli [54], and Pnueli [61].

1.2.2 Related and complementary work

Parallel programming models. Programming models similar in spirit to ours have been proposed by Valiant [73] and Thornley [71]; our model differs in that we provide a more explicit supporting theoretical framework and in the use we make of archetypes.

Automatic parallelization of sequential programs. Our work is in many respects complementary to efforts to develop parallelizing compilers, for example Fortran D [28] and HPF [43]. The focus of such work is on the automatic detection of exploitable parallelism, while our work addresses how to exploit parallelism once it is known to exist. Our theoretical framework could be used to prove not only manually-applied transformations but also those applied by parallelizing compilers.

Programming skeletons, design patterns, and distributed objects. Our work is also in some respects complementary to work exploring the use of programming skeletons and patterns in

parallel computing, for example that of Cole [27] and Brinch Hansen [15], and even work exploring distributed objects, pC++ [12] for example. We also make use of abstractions that capture exploitable commonalities among programs, but we use these abstractions to guide a program development methodology based on program transformations.

Communication libraries. Much work has been done in developing program libraries intended to insulate application developers from the details of the parallel architecture on which their programs are to execute, for example MPI [58]. Our work is complementary to this work in that our archetype-based libraries of communication operations can be implemented using subsets of these more general libraries, and in addition to the libraries we provide strategies for their use.

1.3 Our programming model and methodology

As suggested earlier, the goal of our work is to provide assistance in developing parallel programs whose specifications are like those usually given for sequential programs, in which the specification describes initial states for which the program must terminate and the relation between initial and final states.

1.3.1 The arb model: parallel composition with sequential semantics

Our primary programming model, which we call the ***arb** model*, is simply the standard sequential model extended to include parallel compositions of groups of program elements whose parallel composition is equivalent to their sequential composition. The name (**arb**) is derived from UC [7] and is intended to connote that such groups of program elements may be interleaved in any arbitrary fashion without changing the result. We define a property we call ***arb**-compatibility*, and we show that if a group of program elements is **arb**-compatible, their parallel composition is semantically equivalent to their sequential composition; we call such compositions ***arb** compositions*. Since **arb**-model programs can be interpreted as sequential programs, the extensive body of tools and techniques applicable to sequential programs is applicable to them. In particular:

- Their correctness can be demonstrated formally by using sequential methods.
- They can be refined by sequential semantics-preserving transformations.
- They can be executed sequentially for testing and debugging.

1.3.2 Transformations from the arb model to practical parallel languages

Because the **arb** composition of **arb**-compatible elements can also be interpreted as parallel composition, **arb**-model programs can be executed as parallel programs. Such programs may not make

effective use of typical parallel architectures, however, so our methodology includes techniques for improving their efficiency while maintaining correctness. We define two subsidiary programming models that abstract key features of two classes of parallel architectures: the ***par** model* for shared-memory (single-address-space) architectures, and the *subset **par** model* for distributed-memory (multiple-address-space) architectures. We then develop semantics-preserving transformations to convert **arb**-model programs into programs in one of these subsidiary models. Intermediate stages in this process are usually **arb**-model programs, so the transformations can make use of sequential refinement techniques, and the programs can be executed sequentially. Finally, we indicate how the **par** model can be mapped to practical programming languages for shared-memory architectures and the subset **par** model to practical programming languages for distributed-memory–message-passing architectures. Together, these groups of transformations provide a semantics-preserving path from the original **arb**-model program to a program in a practical programming language.

Figure 1.1 illustrates this overall scheme. Solid-bordered boxes indicate programs in the various models; arrows indicate semantics-preserving transformations. A dashed arrow runs from the box denoting a sequential program to a box denoting an **arb**-model programs because it is sometimes appropriate and feasible to derive an **arb**-model program from an existing sequential program (by replacing sequential compositions of **arb**-compatible elements with **arb** compositions of the same elements).

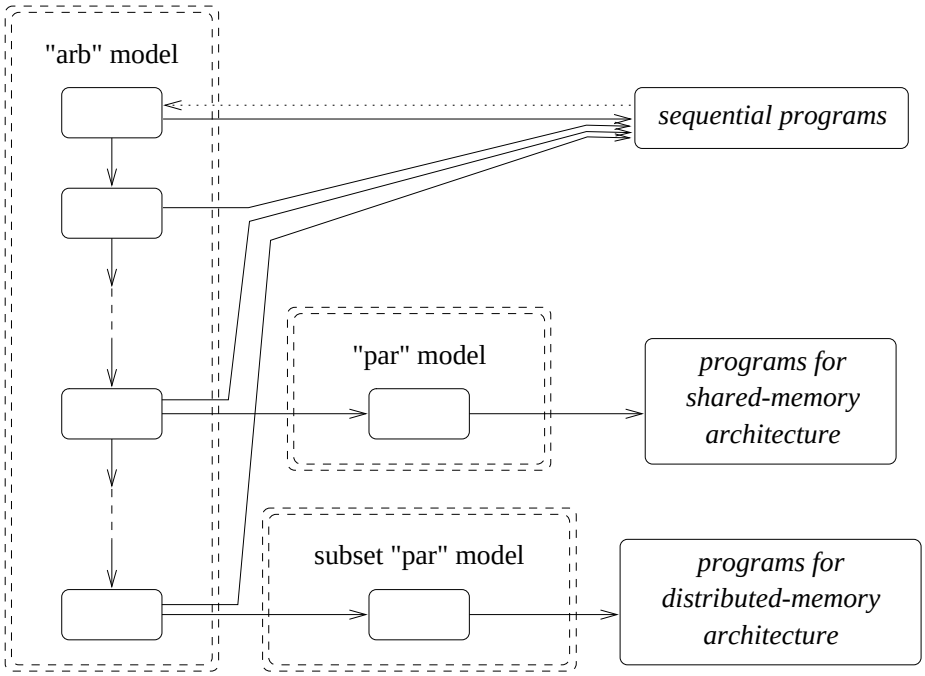


Figure 1.1: Overview of programming models and transformation process.

1.3.3 Supporting framework for proving transformations correct

Some of the transformations indicated in Figure 1.1 — those within the **arb** model — can be proved correct using the techniques of sequential stepwise refinement. Others — those between our different programming models, or from one of our models to a practical programming language — require a different approach. We therefore define an operational model based on viewing programs as state-transition systems, give definitions of our programming models in terms of this underlying operational model, and use it to prove the correctness of those transformations for which sequential techniques are inappropriate.

1.3.4 Programming archetypes

An additional important element of our approach is that we envision the transformation process just described as being guided by what we call *parallel programming archetypes*. An archetype is an abstraction that captures the commonality of a class of programs with common computational structure (e.g., the familiar divide-and-conquer of sequential programming); a parallel programming archetype is such an abstraction for a class of programs whose common features have to do with their parallel structure (e.g., patterns of interprocess communication). We envision application developers choosing from a range of archetypes, each representing a class of programs with common features and providing a class-specific parallelization strategy — that is, a pattern for the shared-memory or distributed-memory program to be ultimately produced — together with a collection of class-specific transformations and a code library of communication or other operations that encapsulate the details of the parallel programs.

1.4 Chapter-by-chapter outline

- Chapter 2 presents our operational model of program semantics and our primary programming model (the **arb** model).
- Chapter 3 presents a collection of useful transformations for **arb** programs. The transformations chosen include most of those required for the example applications in subsequent chapters.
- Chapter 4 presents our approach to transforming **arb** programs into programs for shared-memory architectures: a programming model (the **par** model) together with transformations from the **arb** model to the **par** model and from the **par** model to languages for shared-memory architectures.

- Chapter 5 presents our approach to transforming **arb** programs into programs for distributed-memory architectures: a programming model (the subset **par** model) together with transformations from the **arb** model to the subset **par** model and from the subset **par** model to languages for distributed-memory architectures.
- Chapter 6 presents extended examples of **arb**-model programs and how they can be transformed.
- Chapter 7 presents experiments focused on the archetypes aspects of our work, in which we defined example archetypes and used them to develop applications.
- Chapter 8 presents experiments focused on the transformation aspects of our work, in which we parallelized applications using a sequence of program transformations, with the key transformation formally justified.
- Chapter 9 surveys related and complementary work.
- Chapter 10 presents conclusions and suggests directions for further research.

Chapter 2

The arb model

As discussed previously in Chapter 1, we are interested in developing and refining parallel programs to meet sequential-style specifications. The heart of our approach is identifying groups of program elements that have the useful property that their parallel composition is semantically equivalent to their sequential composition. We call such a group of program elements **arb**-compatible.[1] We can then employ the following approach to program development:

- Write down the program using sequential constructors and parallel composition (||), but ensuring that all groups of elements composed in parallel are **arb**-compatible. We call such a program an ***arb**-model program*, and it can be interpreted as either a sequential program or a parallel program, with identical meaning.
- View the program as a sequential program and operate on it with sequential refinement techniques, which are well-defined and well-understood. In refining a sequential composition whose elements are **arb**-compatible, take care to preserve their **arb**-compatibility. The result is a program that refines the original program and can also be interpreted as either a sequential or a parallel program, with identical meaning.

In this chapter, we first present our operational model for parallel programs, the model we will use for reasoning about programs and program transformations that are not amenable to strictly sequential reasoning techniques. We then define a notion of **arb**-compatibility, such that the parallel composition of a group of **arb**-compatible program elements is semantically equivalent to its sequential composition. We then identify restrictions on groups of program elements that are sufficient to guarantee their **arb**-compatibility, and we present some properties of parallel compositions of **arb**-compatible elements. We give two presentations of this material: one relying mostly on natural

[1] As mentioned in Chapter 1, the name (**arb**) is derived from UC [7] and is intended to connote that such groups of program elements may be interleaved in any arbitrary fashion without changing the result.

language and omitting detailed proofs (Section 2.1 and Section 2.2), and one making more extensive use of symbolic notation and presenting more detailed proofs (Section 2.7 and Section 2.8).

We then revisit these ideas in the context of two representative programming notations: (1) a theory-oriented notation, Dijkstra's guarded-command language [35, 37], where by "theory-oriented" we mean a notation used primarily as a basis for formal work on program semantics, and (2) a practical programming notation, Fortran 90 [1, 46], where by "practical programming notation" we mean a notation used primarily for the development of applications, particularly large-scale ones. We present our ideas in the context of a theory-oriented notation to demonstrate that they can be made rigorous in a notation for which a formal semantics is defined. We present our ideas in the context of a practical programming notation to show that this rigor carries over into the realm of large-scale application development, at least insofar as the practical notation matches the simpler theory-oriented notation. Finally, we show the syntactic transformations necessary to execute **arb**-model programs sequentially or in parallel.

2.1 Program semantics and operational model

We define programs in such a way that a program describes a state-transition system, and show how to define program computations, sequential and parallel composition, and program refinement in terms of this definition. This section presents the material with a minimum of mathematical notation and only brief sketches of most proofs; Section 2.7 presents the same material formally and in more detail, including a description (Section 2.7.1) of notational conventions.

2.1.1 Overview

Treating programs as state-transition systems is not a new approach; it has been used in work such as Chandy and Misra [24], Lynch and Tuttle [52], Lamport [51], Manna and Pnueli [54], and Pnueli [61] to reason about both parallel and sequential programs. The basic notions of a state-transition system — a set of states together with a set of transitions between them, representable as a directed graph with states for vertices and transitions for edges — are perhaps more helpful in reasoning about parallel programs, particularly when program specifications describe ongoing behavior (e.g., safety and progress properties) rather than relations between initial and final states, but they are also applicable to sequential programs. Our operational model builds on this basic view of program execution, presented in a way specifically aimed at facilitating the stating and proving of the main theorems of this chapter (that for groups of program elements meeting stated criteria, their parallel and sequential compositions are semantically equivalent) and subsequent chapters.

Thus, we define programs in terms of sets of variables and sets of program actions. A program's variables define a set of states, one for each assignment of values to variables; the variables can

include not only the "visible" variables of imperative programming languages but also "hidden" variables such as program counters. A program action is defined as a relation between its input variables and its output variables; each program action generates a set of state transitions. Program actions are atomic. The system can be viewed as a graph, with each state a vertex and each state transition a directed edge. A computation of the program then corresponds to a path in the graph, starting from one of the program's initial states and — if the computation terminates — ending in a state with no outgoing edges.

2.1.2 Definitions

Definition 2.1 (Program).

We define a program P as a 6-tuple $(V, L, InitL, A, PV, PA)$, where

- V is a finite set of typed variables. V defines a state space in the state-transition system; that is, a state is given by the values of the variables in V. In our semantics, distinct program variables denote distinct atomic data objects; aliasing is not allowed.
- $L \subseteq V$ represents the *local variables* of P. These variables are distinguished from the other variables of P in two ways: (1) The initial states of P are given in terms of their values, and (2) they are invisible outside P — that is, they may not appear in a specification for P, and they may not be accessed by other programs composed with P, either in sequence or in parallel.
- $InitL$ is an assignment of values to the variables of L, representing their initial values.
- A is a finite set of *program actions*. A program action describes a relation between states of its input variables (those variables in V that affect its behavior, either in the sense of determining from which states it can be executed or in the sense of determining the effects of its execution) and states of its output variables (those variables whose value can be affected by its execution). Thus, a program action is a triple (I_a, O_a, R_a) in which
  - $I_a \subseteq V$ represents the input variables of A.
  - $O_a \subseteq V$ represents the output variables of A.
  - R_a is a relation between I_a-tuples and O_a-tuples.
- $PV \subseteq V$ are *protocol variables* that can be modified only by *protocol actions* (elements of PA). (That is, if v is a protocol variable, and $a = (I_a, O_a, R_a)$ is an action such that $v \in O_a$, a must be a protocol action.) Such variables and actions are not needed in this chapter but are useful in defining the synchronization mechanisms of Chapter 4 and Chapter 5; the requirement that

protocol variables be modified only by protocol actions simplifies the task of defining such mechanisms. Observe that variables in PV can include both local and non-local variables.

- $PA \subseteq A$ are *protocol actions*. Only protocol actions may modify protocol variables. (Protocol actions may, however, modify non-protocol variables.)

□

Remarks about Definition 2.1.

- Program action $a = (I_a, O_a, R_a)$ defines a set of state transitions, each of which we write in the form $s \xrightarrow{a} s'$, as follows: $s \xrightarrow{a} s'$ if the pair (i, o), where i is a tuple representing the values of the variables in I_a in state s and o is a tuple representing the values of the variables in O_a in state s', is an element of relation R_a.
- We can also define a program action based on its set of state transitions, by inferring the required I_a, O_a, and R_a. Details are given in the remarks following Definition 2.1′ in Section 2.7.2.

□

Examples of Definition 2.1.

- As an example, consider the definition of program *skip* (Definition 2.29) in Section 2.9: The program has a single variable, En_{skip}, with an initial value of *true*, and a single action that maps the state s in which En_{skip} is *true* to the state s' in which En_{skip} is *false*. All of the commands and constructs we define have an analogous "enabling" variable, which is *true* exactly when the command or construct is enabled — that is, allowed to begin execution.
- Other simple examples include the remaining commands of Section 2.9.1. Observe that *abort* is unusual in that it never sets its enabling flag to *false* and hence (as intended) never terminates.

□

Definition 2.2 (Initial states).

For program P, s is an *initial state* of P if, in s, the values of the local variables of P have the values given in *InitL*.

□

Definition 2.3 (Enabled).

For action a and state s of program P, we say that a is *enabled* in s exactly when there exists program state s' such that $s \xrightarrow{a} s'$.

□

Remarks about Definition 2.3.

- If we view the program's state-transition system as a graph, a program action is enabled in state s if there is an outgoing edge corresponding to the action from the vertex corresponding to s.

□

Definition 2.4 (Computation).

If $P = (V, L, InitL, A, PV, PA)$, a *computation* of P is a pair

$$C = (s_0, \langle j : 1 \leq j \leq N : (a_j, s_j) \rangle)$$

in which

- s_0 is an initial state of P.
- $\langle j : 1 \leq j \leq N : (a_j, s_j) \rangle$ is a sequence of pairs in which each a_j is a program action of P, and for all j, $s_{j-1} \xrightarrow{a_j} s_j$. We call these pairs the state transitions of C, and the sequence of actions a_j the actions of C.

  N can be a non-negative integer or ∞. In the former case, we say that C is a finite or terminating computation with length $N + 1$ and final state s_N. In the latter case, we say that C is an infinite or nonterminating computation.

- If C is infinite, the sequence $\langle j : 1 \leq j : (a_j, s_j) \rangle$ satisfies the following fairness requirement:

  If, for some state s_j and program action a, a is enabled in s_j, then eventually either a occurs in C or a ceases to be enabled.

□

Remarks about Definition 2.4.

- As noted earlier, if we view the program's state-transition system as a graph, a computation corresponds to a path through the graph, following directed edges (actions) between vertices (states).

□

Definition 2.5 (Terminal state).

We say that state s of program P is a *terminal state* of P exactly when there are no actions of P enabled in s.

□

Remarks about Definition 2.5.

- If we view the program's state-transition system as a graph, a terminal state is one with no outgoing edges.

□

Definition 2.6 (Maximal computation).

We say that a computation of C of P is a *maximal computation* exactly when either (1) C is infinite or (2) C is finite and ends in a terminal state.

□

Definition 2.7 (Affects).

For predicate q and variable $v \in V$, we say that v *affects* q exactly when there exist states s and s', identical except for the value of v, such that $q.s \neq q.s'$.

For expression E and variable $v \in V$, we say that v affects E exactly when there exists value k for E such that v affects the predicate $(E = k)$.

□

2.1.3 Specifications and program refinement

The usual meaning of "program P is refined by program P'" is that program P' meets any specification met by P. We will confine ourselves to specifications that describe a program's behavior in terms of initial and final states, giving (1) the set of initial states s such that the program is guaranteed to terminate if started in s, and (2) the relation, for terminating computations, between initial and final states. An example of such a specification is a Hoare total-correctness triple. In terms of our model, initial and final states correspond to assignments of values to the program's variables; we make the additional restriction that specifications do not mention a program's local variables L. We make this restriction because otherwise program equivalence can depend on internal behavior (as reflected in the values of local variables), which is not the intended meaning of equivalence.[2] We write $P \sqsubseteq P'$ to denote that P is refined by P'; if $P \sqsubseteq P'$ and $P' \sqsubseteq P$, we say that P and P' are equivalent, and write $P \sim P'$.

Definition 2.8 (Equivalence of computations).

For programs P_1 and P_2 and a set of typed variables V such that $V \subseteq V_1$ and $V \subseteq V_2$ and for every v in V, v has the same type in all three sets (V, V_1, and V_2), we say that computations C_1 of P_1 and C_2 of P_2 are *equivalent with respect to* V exactly when:

- For every v in V, the value of v in the initial state of C_1 is the same as its value in the initial state of C_2.
- Either (1) both C_1 and C_2 are infinite, or (2) both are finite, and for every v in V, the value of v in the final state of C_1 is the same as its value in the final state of C_2.

□

We can now give a sufficient condition for showing that $P_1 \sqsubseteq P_2$ in our semantics.

Theorem 2.9 (Refinement in terms of equivalent computations).

For P_1 and P_2 with $(V_1 \setminus L_1) \subseteq (V_2 \setminus L_2)$ (where $\setminus$ denotes set difference), $P_1 \sqsubseteq P_2$ when for every maximal computation C_2 of P_2 there is a maximal computation C_1 of P_1 such that C_1 is equivalent to C_2 with respect to $(V_1 \setminus L_1)$.

□

[2] For example, if specifications were allowed to mention local variables, sequential and parallel composition would not be associative, since different ways of parenthesizing the composition lead to different sets of local variables.

Proof of Theorem 2.9.

This follows immediately from Definition 2.8, the usual definition of refinement, and our restriction that program specifications not mention local variables.

□

2.1.4 Program composition

We now present definitions of sequential and parallel composition in terms of our model. First we need some restrictions to ensure that the programs to be composed are compatible — that is, that it makes sense to compose them:

Definition 2.10 (Composability of programs).

We say that a set of programs $P_1, \ldots, P_N$ *can be composed* exactly when

- any variable that appears in more than one program has the same type in all the programs in which it appears (and if it is a protocol variable in one program, it is a protocol variable in all programs in which it appears),
- any action that appears in more than one program is defined in the same way in all the programs in which it appears, and
- different programs do not have local variables in common.

□

Remarks about Definition 2.10.

- If it should be the case that for some $j \neq k$, the local variables of P_j and P_k overlap, observe that we can rename (in P_j or P_k) any variable v in both sets of local variables without changing the meaning of the modified program.

□

2.1.4.1 Sequential composition

The usual meaning of sequential composition is this: A maximal computation of $P_1; P_2$ is a maximal computation C_1 of P_1 followed (if C_1 is finite) by a maximal computation C_2 of P_2, with the obvious generalization to more than two programs. We can give a definition with this meaning in terms of our model by introducing additional local variables $En_1, \ldots, En_N$ that ensure that things happen in the proper sequence, as follows: Actions from program P_j can execute only when En_j is *true*. En_1 is set to *true* at the start of the computation, and then as each P_j terminates it sets En_j to *false* and En_{j+1} to *true*, thus ensuring the desired behavior.

Definition 2.11 (Sequential composition).

If programs $P_1, \ldots, P_N$, with $P_j = (V_j, L_j, InitL_j, A_j, PV_j, PA_j)$, can be composed (Definition 2.10), we define their sequential composition $(P_1; \ldots; P_N) = (V, L, InitL, A, PA, PV)$ thus:

- $V = V_1 \cup \ldots \cup V_N \cup L$.
- $L = L_1 \cup \ldots \cup L_N \cup \{En_P, En_1, \ldots, En_N\}$, where $En_P, En_1, \ldots, En_N$ are distinct Boolean variables not otherwise occurring in V:

  En_P is *true* in the initial state of the sequential composition and *false* thereafter.

  For all j, En_j is *true* during (and only during) the part of the computation corresponding to execution of P_j.
- $InitL$ is defined thus: The initial value of En_P is *true*. For all j, the initial value of En_j is *false*, and the initial values of variables in L_j are those given by $InitL_j$.
- A consists of the following types of actions:
  - Actions corresponding to actions in A_j, for some j: For $a \in A_j$, we define a' identical to a except that a' is enabled only when $En_j = true$.
  - Actions that accomplish the transitions between components of the composition:

    Initial action a_{T_0} takes any initial state s, with $En_P = true$, to a state s' identical except that $En_P = false$ and $En_1 = true$. s' is thus an initial state of P_1.

    For j with $1 \leq j < N$, action a_{T_j} takes any terminal state s of P_j, with $En_j = true$, to a state s' identical except that $En_j = false$ and $En_{j+1} = true$. s' is thus an initial state of P_{j+1}.

    Final action a_{T_N} takes any terminal state s of P_N, with $En_N = true$, to a state s' identical except that $En_N = false$. s' is thus a terminal state of the sequential composition.
- $PV = PV_1 \cup \ldots \cup PV_N$.

- PA contains exactly those actions a' derived (as described above) from the actions a of $PA_1 \cup \ldots \cup PA_N$.

□

Remarks about Definition 2.11.

- Sequential composition as just defined is associative, since our definition of program equivalence (two-sided refinement) excludes local variables.

□

2.1.4.2 Parallel composition

The usual meaning of parallel composition is this: A computation of $P_1||P_2$ defines two threads of control, one each for P_1 and P_2. Initiating the composition corresponds to starting both threads; execution of the composition corresponds to an interleaving of actions from both components; and the composition is understood to terminate when both components have terminated. We can give a definition with this meaning in terms of our model by introducing additional local variables that ensure that the composition terminates when all of its components terminate, as follows: As for sequential composition, we introduce additional local variables $En_1, \ldots, En_N$ such that actions from program P_j can execute only when En_j is *true*. For parallel composition, however, all of the En_j's are set to *true* at the start of the computation, so computation is an interleaving of actions from the P_j's. As each P_j terminates, it sets the corresponding En_j to *false*; when all are *false*, the composition has terminated. Observe that the definitions of parallel and sequential composition are almost identical; this greatly facilitates the proofs of Lemma 2.17 and Lemma 2.18.

Definition 2.12 (Parallel composition).

If programs $P_1, \ldots, P_N$, with $P_j = (V_j, L_j, InitL_j, A_j, PV_j, PA_j)$, can be composed (Definition 2.10), we define their parallel composition $(P_1|| \ldots ||P_N) = (V, L, InitL, A, PV, PA)$ thus:

- $V = V_1 \cup \ldots \cup V_N \cup L$.
- $L = L_1 \cup \ldots \cup L_N \cup \{En_P, En_1, \ldots, En_N\}$, where $En_P, En_1, \ldots, En_N$ are distinct Boolean variables not otherwise occurring in V.

  En_P is *true* in the initial state of the parallel composition and *false* thereafter.

  For all j, En_j is *true* until the part of the composition corresponding to P_j has terminated.

- *InitL* is defined thus: The initial value of En_P is *true*. For all j, the initial value of En_j is *false*, and the initial values of variables in L_j are those given by $InitL_j$.
- A consists of the following types of actions:
  - Actions corresponding to actions in A_j, for some j: For $a \in A_j$, we define a' identical to a except that a' is enabled only when En_j is *true*.
  - Actions that correspond to the initiation and termination of the components of the composition:

    Initial action a_{T_0} takes any initial state s, with $En_P = true$, to a state s' identical except that $En_j = true$ for all j. s' is thus an initial state of P_j, for all j.

    For j with $1 \leq j \leq N$, action a_{T_j} takes any terminal state s of P_j, with $En_j = true$, to a state s' identical except that $En_j = false$. A terminating computation of P contains one execution of each a_{T_j}; after execution of a_{T_j} for all j, the resulting state s' is a terminal state of the parallel composition.
- $PV = PV_1 \cup \ldots \cup PV_N$.
- PA contains exactly those actions a' derived (as described above) from the actions a of $PA_1 \cup \ldots \cup PA_N$.

□

Remarks about Definition 2.12.

- Parallel composition as just defined is associative and commutative, since our definition of program equivalence (two-sided refinement) excludes local variables.

□

2.2 arb-compatibility and arb composition

We now turn our attention to defining sufficient conditions for a group of programs $P_1, \ldots, P_N$ to have the property we want, namely:

$$(P_1 || \ldots || P_N) \sim (P_1; \ldots; P_N) \quad .$$

This section presents the material with a minimum of mathematical notation and only brief sketches of most proofs; Section 2.8 presents the same material in more detail, with more complete proofs and with more use of mathematical notation.

2.2.1 Definition of arb-compatibility

We first define a key property of pairs of program actions.

Definition 2.13 (Commutativity of actions).

Actions a and b of program P are said to *commute* exactly when the following two conditions hold:

- Execution of b does not affect (in the sense of Definition 2.7) whether a is enabled, and vice versa.
- It is possible to reach s_2 from s_1 by first executing a and then executing b exactly when it is also possible to reach s_2 from s_1 by first executing b and then executing a, as illustrated in Figure 2.1. In the figure, a and b are both nondeterministic, but observe that the graph has the property that if we can reach a state (s_2 or s_2') by executing first a and then b, then we can reach the same state by first executing b and then a, and vice versa.

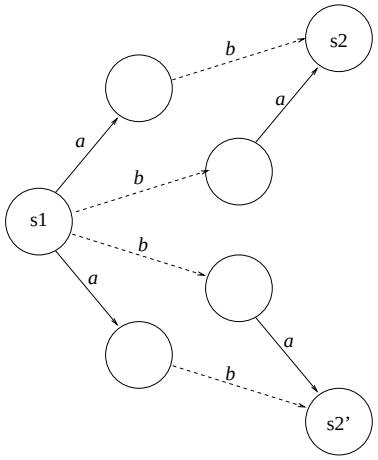


Figure 2.1: Commutativity of actions a and b.

□

Remarks about Definition 2.13.

- a and b commute exactly when a and b have the diamond property [25, 53].

□

We now define the desired condition.

Definition 2.14 (arb-compatible).

Programs $P_1, \ldots, P_N$ are ***arb**-compatible* exactly when they can be composed (Definition 2.10) and any action in one program commutes (Definition 2.13) with any action in another program.

□

2.2.2 Equivalence of sequential and parallel composition for arb-compatible components

We now show that **arb**-compatibility guarantees the property of interest, namely the equivalence of parallel and sequential composition. We sketch the proof here; a detailed proof is given in Section 2.8.2.

Theorem 2.15 (Parallel $\sim$ sequential for arb-compatible programs).

If $P_1, \ldots, P_N$ are **arb**-compatible, then

$$(P_1 || \ldots || P_N) \sim (P_1; \ldots; P_N) \ .$$

□

Proof of Theorem 2.15.

We write $P_P = (P_1 || \ldots || P_N)$ and $P_S = (P_1; \ldots; P_N)$. From Definition 2.11 and Definition 2.12,

$$(V_P = V_S) \wedge (L_P = L_S) \wedge (InitL_P = InitL_S) \wedge (PV_P = PV_S) \wedge (PA_P = PA_S) \ ,$$

so we write $P_P = (V, L, InitL, A_P, PV, PA)$ and $P_S = (V, L, InitL, A_S, PV, PA)$. We proceed as follows:

- We first show (Lemma 2.17) that for every maximal computation C_S of P_S there is a maximal computation C_P of P_P with C_S equivalent to C_P with respect to $V \setminus L$. From Theorem 2.9, this establishes that $P_P \sqsubseteq P_S$.
- We then show (Lemma 2.18) the converse: that for every maximal computation C_P of P_P there is a maximal computation C_S of P_S with C_P equivalent to C_S with respect to $V \setminus L$. From Theorem 2.9, this establishes that $P_S \sqsubseteq P_P$.
- We then conclude that $P_P \sim P_S$, as desired.

□

We begin by proving the following useful lemma:

Lemma 2.16 (Reordering of computations).

Suppose that $P_1, \ldots, P_N$ are **arb**-compatible and C_P is a finite (not necessarily maximal) computation of $P_P = (P_1 || \ldots || P_N)$ containing a successive pair of transitions $((a, s_n), (b, s_{n+1}))$ such that a and b commute. Then we can construct a computation C'_P of P_P with the same initial and final states as C_P, and the same sequence of transitions, except that the pair $((a, s_n), (b, s_{n+1}))$ has been replaced by the pair $((b, s'_n), (a, s_{n+1}))$.

□

Proof of Lemma 2.16.

This is an obvious consequence of the commutativity (Definition 2.13) of a and b: If $s_{n-1} \xrightarrow{a} s_n$ and $s_n \xrightarrow{b} s_{n+1}$, then there exists a state s'_n such that $s_{n-1} \xrightarrow{b} s'_n$ and $s'_n \xrightarrow{a} s_{n+1}$, so we can construct a computation as described.

□

Lemma 2.17 (Sequential refines parallel).

For P_P and P_S defined as in Theorem 2.15, if C_S is a maximal computation of P_S, there is a maximal computation C_P of P_P with C_S equivalent to C_P with respect to $V \setminus L$.

□

Proof of Lemma 2.17.

The proof of this lemma is straightforward for finite computations: We have defined parallel and sequential composition in such a way that any maximal finite computation of the parallel computation maps to an equivalent maximal computation of the parallel composition.

For nonterminating computations, we can similarly map a computation of the sequential composition to an infinite sequence of transitions of the parallel composition. However, the result may not be a computation of the parallel composition because it may violate the fairness requirement: If P_j fails to terminate, no action of P_{j+1} can occur, even though in the parallel composition there may be actions of P_{j+1} that are enabled. If this is the case, however, we can use the principle behind Lemma 2.16 to transform the unfair sequence of transitions into a fair one.

Details are given in the proof of Lemma 2.17′ in Section 2.8.2.

□

Lemma 2.18 (Parallel refines sequential).

For P_P and P_S defined as in Theorem 2.15, if C_P is a maximal computation of P_P, there is a maximal computation C_S of P_S such that C_S is equivalent to C_P with respect to $V \setminus L$.

□

Proof of Lemma 2.18.

For terminating computations, the proof is straightforward: Given a maximal computation of the parallel composition, we first apply Lemma 2.16 repeatedly to construct an equivalent (also maximal) computation of the parallel composition in which, for $j < k$, all transitions corresponding to actions of P_j occur before transitions corresponding to actions of P_k. As in the proof of Lemma 2.17, this computation then maps to an equivalent maximal computation of the sequential composition.

For nonterminating computations, we can once again use the principle behind Lemma 2.16 to construct a sequence of transitions (of the parallel composition) in which, for $j < k$, all transitions corresponding to actions of P_j occur before transitions corresponding to actions of P_k. We then map this sequence of transitions to a computation of the sequential composition.

Details are given in the proof of Lemma 2.18′ in Section 2.8.2.

□

2.2.3 Definition of arb composition

For **arb**-compatible programs $P_1, \ldots, P_N$, then, we know that

$$(P_1 || \ldots || P_N) \sim (P_1; \ldots; P_N) \ .$$

To denote this parallel/sequential composition of **arb**-compatible elements, we write $\mathbf{arb}(P_1, \ldots, P_N)$, where

$$\mathbf{arb}(P_1, \ldots, P_N) \sim (P_1 || \ldots || P_N)$$

or equivalently

$$\mathbf{arb}(P_1, \ldots, P_N) \sim (P_1; \ldots; P_N) \ .$$

We refer to this notation as "**arb** composition", although it is not a true composition operator since it is properly applied only to groups of elements that are **arb**-compatible. We regard it as a useful

form of syntactic sugar that denotes not only the parallel/sequential composition of $P_1, \ldots, P_N$ but also the fact that $P_1, \ldots, P_N$ are **arb**-compatible.

We also define an additional bit of syntactic sugar, $\mathbf{seq}(P_1, \ldots, P_N)$, such that

$$\mathbf{seq}(P_1, \ldots, P_N) \sim (P_1; \ldots; P_N) \ .$$

We will sometimes use this notation to improve the readability of nestings of sequential and **arb** composition.

2.2.4 Properties of arb composition

arb composition has the following properties.

Theorem 2.19 (Associativity of arb composition).

arb composition is associative.

□

Proof of Theorem 2.19.

We must show that if P_1, P_2, P_3 are **arb**-compatible, then

$$\mathbf{arb}(P_1, \mathbf{arb}(P_2, P_3)) \sim \mathbf{arb}(\mathbf{arb}(P_1, P_2), P_3) \ .$$

This theorem is an obvious consequence of the definition of **arb** composition (Definition 2.14) and the associativity of parallel composition, except that it is not immediately obvious that the two sides of the claimed equivalence make sense. Recalling that we only write $\mathbf{arb}(P_1, \ldots, P_N)$ when $P_1, \ldots, P_N$ are **arb**-compatible, the equivalence makes sense only if:

- P_1 and P_2 are **arb**-compatible, as are P_2 and P_3.
- P_1 and $\mathbf{arb}(P_2, P_3)$ are **arb**-compatible, as are P_3 and $\mathbf{arb}(P_1, P_2)$.

The former is an obvious corollary of the definition of **arb**-compatibility (Definition 2.14): If $P_1, \ldots, P_N$ are **arb**-compatible, then clearly the elements of any subset of $P_1, \ldots, P_N$ are **arb**-compatible as well. The latter follows from the definitions of **arb**-compatibility and parallel composition (Definition 2.12): $\mathbf{arb}(P_2, P_3) \sim (P_2 || P_3)$, and then from the definitions it is clear that $(P_2 || P3)$ and P_1 are **arb**-compatible. The case of $\mathbf{arb}(P_1, P_2)$ and P_3 is exactly analogous. So now we can proceed[3]:

[3]For an explanation of this calculational proof style, refer to Section 2.7.1

$\mathbf{arb}(P_1, \mathbf{arb}(P_2, P_3))$

$\sim$ { definitions }

$(P_1 || (P_2 || P_3))$

$\sim$ { associativity of parallel composition }

$((P_1 || P_2) || P_3)$

$\sim$ { definitions }

$\mathbf{arb}(\mathbf{arb}(P_1, P_2), P_3)$

□

Theorem 2.20 (Commutativity of arb composition).

arb composition is commutative.

□

Proof of Theorem 2.20.

This follows directly from the equivalence of **arb** and parallel composition and the commutativity of parallel composition.

□

Theorem 2.21 (Refinement by parts of arb composition).

We can refine any component of an **arb** composition to obtain a refinement of the whole composition. That is, if $P_1, \ldots, P_N$ are **arb**-compatible, and, for each j, $P_j \sqsubseteq P'_j$, and $P'_1, \ldots, P'_N$ are **arb**-compatible, then

$\mathbf{arb}(P_1, \ldots, P_N)$

$\sqsubseteq$

$\mathbf{arb}(P'_1, \ldots, P'_N)$

□

Proof of Theorem 2.21.

$\mathbf{arb}(P_1, \ldots, P_N)$

$\sim$ { Theorem 2.15 }

$(P_1; \ldots; P_N)$

$\sqsubseteq$ { refinement by parts for sequential programs }

$(P'_1; \ldots; P'_N)$

$\sim$ { Theorem 2.15 and hypothesis }

$\mathbf{arb}(P'_1, \ldots, P'_N)$

□

Remarks about Theorem 2.21.

- This theorem, as mentioned earlier, is the justification for our program-development strategy, in which we apply the techniques of sequential stepwise refinement to **arb**-model programs.

□

2.2.5 A simpler sufficient condition for arb-compatibility

The definition of **arb**-compatibility given in Definition 2.14 is the most general one that seems to give the desired properties (equivalence of parallel and sequential composition, and associativity and commutativity), but it may be difficult to apply in practice. We therefore give a more-easily-checked sufficient condition for programs $P_1, \ldots, P_N$ to be **arb**-compatible.

First, we give some preliminary definitions:

Definition 2.22 (Variables read by P).

For program P, we say that a variable v is *read by* P if it is an input variable for some action a of P; we write VR to denote the set of all such variables.

□

Definition 2.23 (Variables written by P).

For program P, we say that a variable v is *written by* P if it is an output variable for some action a of P; we write VW to denote the set of all such variables.

□

We can now give the sufficient condition, preceded by a preliminary definition.

Definition 2.24 (Programs that share only read-only variables).

If programs $P_1, \ldots, P_N$ can be composed (Definition 2.10), and for $j \neq k$, no variable written by P_j is read or written by P_k, then we say that $P_1, \ldots, P_N$ *share only read-only variables.*

□

Theorem 2.25 (arb-compatibility and shared variables).

If programs $P_1, \ldots, P_N$ share only read-only variables (Definition 2.24), then $P_1, \ldots, P_N$ are **arb**-compatible.

□

Proof of Theorem 2.25.

Given programs $P_1, \ldots, P_N$ that satisfy the condition, it suffices to show that any two actions from distinct components P_j and P_k commute. The proof is straightforward; a detailed version appears as the proof of Theorem 2.25′ in Section 2.8.

□

2.3 arb composition and programming notations

A key difficulty in applying our methodology for program development is in identifying groups of program elements that are known to be **arb**-compatible. The difficulty is exacerbated by the fact that many programming notations have a notion of program variable that is more difficult to work with than the notion we employ for our formal semantics. In our semantics, variables with distinct names address distinct data objects. In many programming notations, this need not be the case, and the difficulty of detecting situations in which variables with distinct names overlap (aliasing) complicates automatic program optimization and parallelization just as it complicates the application of our methodology. Syntactic restrictions sufficient to guarantee **arb**-compatibility do not seem in general feasible, but we believe that the semantic restrictions described in this section are a step in the right direction, by helping programmers to make conservative estimates of which variables are being accessed, that is, to identify a superset of the variables being accessed.

Our approach is to define, for every program P, sets of variables **ref**.P and **mod**.P, such that **ref**.$P \supseteq VR_P$ and **mod**.$P \supseteq VW_P$. That is, **mod**.P contains all atomic data objects[4] whose value is changed in some computation of P, and **ref**.P contains all atomic data objects referenced in P, that is, all data objects whose value is "read" during some computation of P. We also define, for every expression E, an analogous **ref**.E such that **ref**.E contains all atomic data objects whose value affects E. Note that it may be the case that **ref**.$P \supset VR_P$ and **mod**.$P \supset VW_P$ — that is, **ref**.P and **mod**.P may be defined more broadly than necessary. Note also that it is not necessarily the case that **mod**.$P \subseteq$ **ref**.P.

We can then state restrictions in terms of **ref** and **mod** sufficient to guarantee **arb**-compatibility:

Theorem 2.26 (arb-compatibility in terms of ref and mod).

Program blocks $P_1, \ldots, P_N$ are **arb**-compatible when, for all $j \neq k$, **mod**.P_j does not intersect **ref**.$P_k \cup$ **mod**.P_k.

□

Proof of Theorem 2.26.

This follows immediately from Theorem 2.25.

□

Remarks about Theorem 2.26.

- It is important to note again that **ref** and **mod** refer to data objects — i.e., memory locations — rather than variable names. In determining which variables to include, users must consider not only questions of aliasing but also the presence of "hidden" variables, examples of which range from the `COMMON`-block variables of Fortran to the hidden variables often involved in file access. (For example, if program P accesses a file sequentially, **mod**.P should include a variable representing the file, since concurrent attempts by two programs to read the same file result in program actions that do not meet the commutativity test for **arb**-compatibility.)

□

[4] An atomic data object is as defined in our semantics or, equivalently, in HPF [43]: one that contains no subobjects — e.g., a scalar data object or a scalar element of an array.

2.4 arb composition and Dijkstra's guarded-command language

2.4.1 Dijkstra's guarded-command language and our model

It is straightforward to define the commands and constructors of Dijkstra's guarded-command language [35, 37] in terms of our model. We sketch such definitions in Section 2.9.

2.4.2 Conditions for arb-compatibility

Giving syntactic restrictions that we know guarantee **arb**-compatibility seems less problematical in Dijkstra's guarded-command language than in a large practical programming language, simply because Dijkstra's guarded-command language is a small and well-understood language. Nevertheless, there is no guarantee that variables with distinct names in fact address distinct data objects, so problems with aliasing are possible. We nonetheless give some examples of defining $\mathbf{mod}.P$ and $\mathbf{ref}.P$ for some of the constructs of Dijkstra's guarded-command language, noting that these examples work only if distinct variable names in fact address distinct data objects.

$\mathbf{mod}.\mathit{skip} = \{\}$

$\mathbf{mod}.\mathit{abort} = \{\}$

$\mathbf{mod}.(x := E) = \{x\}$

$(P = s_1; \ldots; s_N) \Rightarrow$

$\quad (\mathbf{mod}.P = (\mathbf{mod}.s_1 \cup \ldots \cup \mathbf{mod}.s_N))$

$(P = \mathbf{if}\ b_1 \rightarrow s_1\ [] \ \ldots\ []\ b_N \rightarrow s_N\ \mathbf{fi}) \Rightarrow$

$\quad (\mathbf{mod}.P = (\mathbf{mod}.s_1 \cup \ldots \cup \mathbf{mod}.s_N)))$

$(P = \mathbf{do}\ b_1 \rightarrow s_1\ [] \ \ldots\ []\ b_N \rightarrow s_N\ \mathbf{od}) \Rightarrow$

$\quad (\mathbf{mod}.P = (\mathbf{mod}.s_1 \cup \ldots \cup \mathbf{mod}.s_N))$

$\mathbf{ref}.E = \{v :: v \text{ is named in } E\}$

$\mathbf{ref}.\mathit{skip} = \{\}$

$\mathbf{ref}.\mathit{abort} = \{\}$

$\mathbf{ref}.(x := E) = \mathbf{ref}.E$

$(P = s_1; \ldots; s_N) \Rightarrow$

$\quad (\mathbf{ref}.P = (\mathbf{ref}.s_1 \cup \ldots \cup \mathbf{ref}.s_N))$

$(P = \mathbf{if}\ b_1 \rightarrow s_1\ [] \ \ldots\ []\ b_N \rightarrow s_N\ \mathbf{fi}) \Rightarrow$

$\quad (\mathbf{ref}.P = (\mathbf{ref}.b_1 \cup \ldots \cup \mathbf{ref}.b_N) \cup (\mathbf{ref}.s_1 \cup \ldots \cup \mathbf{ref}.s_N))$

$(P = \mathbf{do}\ b_1 \rightarrow s_1\ [] \ \ldots\ []\ b_N \rightarrow s_N\ \mathbf{od}) \Rightarrow$

$\quad (\mathbf{ref}.P = (\mathbf{ref}.b_1 \cup \ldots \cup \mathbf{ref}.b_N) \cup (\mathbf{ref}.s_1 \cup \ldots \cup \mathbf{ref}.s_N))$

2.4.3 Examples of arb composition

Composition of assignments

This example composes two simple assignment commands.

$$\mathbf{arb}(a := 1, b := 2)$$

Composition of sequential blocks

This example composes two sequences, the first assigning to a and b and the second assigning to c and d.

$$\mathbf{arb}(\mathbf{seq}(a := 1, b := a), \mathbf{seq}(c := 2, d := c))$$

Invalid composition

The following example is not a valid **arb** composition; the two assignments are not **arb**-compatible.

$$\mathbf{arb}(a := 1, b := a)$$

2.5 arb composition and Fortran 90

2.5.1 Fortran 90 and our model

Giving a formal definition of the semantics of a large practical programming language such as Fortran 90 [1, 46] is far from trivial. We observe, however, that the well-understood constructs of Dijkstra's guarded-command language have, when deterministic, analogous constructs in Fortran 90 (as in many other practical languages), and that formally-justified results derived in Dijkstra's guarded-command language apply to Fortran 90 programs insofar as the Fortran 90 programs limit themselves to these analogous constructs. Difficulties in applying our work to Fortran 90 fall into two categories:

Irregular control structures. Giving a formal definition of the semantics of less-than-disciplined control structures such as `GOTO` is troublesome but possible. However, our definitions of sequential and parallel composition make sense only for self-contained program blocks, where a self-contained block is one that contains neither a `GOTO` whose target lies outside the block nor a label that is the target of a `GOTO` outside the block. We observe that our results on **arb** composition apply to compositions of self-contained blocks; we do not attempt to give a meaning for composition of blocks that are not self-contained.

Aliased and hidden variables. Fortran, particularly FORTRAN 77, is notorious for making it difficult to determine from the program text which variables are accessed or modified: Variables with different names may reference the same location (aliasing, as a result of `EQUIVALENCE` statements or of the use of different array indices with the same value), and references to `COMMON`-block and other "hidden" variables may be difficult to determine without interprocedural analysis. Our results on **arb** composition apply provided it is known *to the programmer* exactly which variables are being addressed in a particular program. Inferring such information (which variables are being addressed) by means of syntactic analysis is not trivial — if it were, parallelizing compilers would be easier to produce — but we believe that it is feasible for programmers to make such determinations manually for programs written in a disciplined style or thoroughly documented.

2.5.2 Conditions for arb-compatibility

As noted in the preceding section, the general problem of determining which variables a program element references and modifies does not seem to be readily amenable to syntactic analysis. We give here some examples of defining **mod**.P and **ref**.P for some example programs.

Simple example

Given the following program block `p`:

```
integer x, y, z
x = y + z
```

we have

$$\textbf{mod}.\mathtt{p} = \{\mathtt{x}\}$$
$$\textbf{ref}.\mathtt{p} = \{\mathtt{y}, \mathtt{z}\}$$

Example with `COMMON` block

Given the following program block `q`:

```
integer u, v
call qsub(u, v)
```

and the following subprogram `qsub`:

```
subroutine qsub(x, y)
integer x, y
common /qcom/ c
integer c
x = 2*y
c = c+1
end subroutine
```

we have:

$\mathbf{mod}.\mathtt{q} = \{\mathtt{u}, \mathtt{c}\}$

$\mathbf{ref}.\mathtt{q} = \{\mathtt{v}, \mathtt{c}\}$

This is an example of a program in which there is no obvious way to determine by syntactic analysis (without interprocedural analysis, which may not be feasible) that a call to a subprogram (`qsub`, called from `q`) modifies a "hidden" variable (`COMMON`-block variable `c`).

2.5.3 Notation

For Fortran 90, we provide a different notation for **arb** composition and explicit sequential composition, one that is analogous to the other constructs of Fortran 90. This notation allows us to write programs in the **arb** model that can be easily, even mechanically, transformed into programs in languages based on Fortran 90, as described in Section 2.6.

2.5.3.1 arb composition

For **arb**-compatible programs $P_1, \ldots, P_N$, we write their **arb** composition thus:

```
arb
    P_1
    P_2
    ...
    P_N
end arb
```

2.5.3.2 seq composition

For any programs $P_1, \ldots, P_N$, we write their sequential composition thus:

```
seq
    P_1
    P_2
    ...
    P_N
end seq
```

Observe that sequential composition is the default; that is, statements are composed sequentially unless they are explicitly composed using parallel composition or one of its restricted forms, **arb** and **par**. (**par** composition is defined in Chapter 4.)

2.5.3.3 arball

To allow us to express the **arb** composition of, for example, the iterations of a loop, we define an indexed form of **arb** composition, with syntax modeled after that of the `FORALL` construct of High Performance Fortran [43], as follows. This notation is syntactic sugar only, and all theorems that apply to **arb** composition apply to **arball** as well.

Definition 2.27 (arball).

If we have N index variables $i_1, \ldots, i_N$, with corresponding index ranges $i_j\_start \leq i_j \leq i_j\_end$, and program block P such that P does not modify the value of any of the index variables — that is, $\mathbf{mod}.P \cap \{i_1, \ldots, i_N\} = \{\}$ — then we can define an **arball** composition as follows:

For each tuple $(x_1, \ldots, x_N)$ in the cross product of the index ranges, we define a corresponding program block $P(x_1, \ldots, x_N)$ by replacing index variables $i_1, \ldots, i_N$ with corresponding values $x_1, \ldots, x_N$. If the resulting program blocks are **arb**-compatible, then we write their **arb** composition as follows:

```
arball (i_1 = i_1_start : i_1_end , ..., i_N = i_N_start : i_N_end)
    P(x_1, ..., x_N)
end arball
```

□

Remarks about Definition 2.27.

- Observe that the body of the **arball** composition can be a sequential composition. We do not require that the sequential composition be explicit, as illustrated in the next-to-last example.

□

2.5.4 Examples of arb composition

Composition of assignments

This example composes two simple assignment commands.

```
arb
    a = 1
    b = 2
end arb
```

Composition of sequential blocks

This example composes two sequences, the first assigning to `a` and `b` and the second assigning to `c` and `d`.

```
arb
    seq
        a = 1
        b = a
    end seq
    seq
        c = 2
        d = c
    end seq
end arb
```

Invalid composition

The following example is not a valid **arb** composition; the two assignments are not **arb**-compatible.

```
arb
    a = 1
    b = a
end arb
```

Invalid composition because of aliasing

The following example is not a valid **arb** composition; because of the `EQUIVALENCE` statement the two assignments are not **arb**-compatible.

```
equivalence (a, b)
arb
    a = 1
    b = 2
end arb
```

Composition of assignments (arball)

The following example composes twenty assignments, one for each pair of values for `i` and `j`:

```
arball (i = 1:4, j = 1:5)
    a(i,j) = i+j
end arball
```

That is, it is equivalent to the following:

```
arb
    a(1,1) = 1+1
    a(2,1) = 2+1
    ...
    a(4,1) = 4+1
    a(1,2) = 1+2
    ...
    a(4,5) = 4+5
end arb
```

Composition of sequential blocks (arball)

The following example composes ten sequences, each assigning to one element of `a` and one element of `b`.

```
arball (i = 1:10)
    seq
        a(i) = i
        b(i) = a(i)
    end seq
end arball
```

As noted in the remarks following Definition 2.27, if the body of the **arball** composition is a sequential composition, we do not require that the sequential composition be explicit; that is, this example could also be written:

```
arball (i = 1:10)
    a(i) = i
    b(i) = a(i)
end arball
```

without changing its meaning.

Invalid composition (arball)

The following example is not a valid **arball**; the ten assignment statements it defines are not **arb**-compatible.

```
arball (i = 1:10)
    a(i+1) = a(i)
end arball
```

2.6 Execution of arb-model programs

Since for **arb**-compatible program elements, their **arb** composition is semantically equivalent to their parallel composition and also to their sequential composition, programs written using sequential commands and constructors plus (valid) **arb** composition can, as noted earlier, be executed either as sequential or as parallel programs with identical results.[5] In this section we discuss how to do this in the context of practical programming languages.

[5] Programs that use **arb** to compose elements that are not **arb**-compatible cannot, of course, be guaranteed to have this property. As discussed in Section 2.2.3, we assume that the **arb** composition notation is applied only to groups of program elements that are **arb**-compatible; it is the responsibility of the programmer to ensure that this is the case.

2.6.1 Sequential execution

A program in the **arb** model can be executed sequentially; such a program can be transformed into an equivalent program in the underlying sequential notation by replacing **arb** composition with sequential composition. For Fortran 90, this is done by removing `arb` and `end arb` and transforming `arball` into nested `DO` loops, as illustrated by the following examples.

Combination of arb and arball

The following program block

```
arb
    arball (i = 2:N-1)
        a(i) = 0
    end arball
    a(1) = 1
    a(N) = 1
end arb
```

is equivalent to the sequential block

```
do i = 2, N-1
    a(i) = 0
end do
a(1) = 1
a(N) = 1
```

Observe that the loop could equally well be executed in reverse order (`do i = N-1, 2, -1`).

arball with multiple indices

The following program block

```
arball (i = 1:N, j = 1:M)
    call p(i, j)
end arball
```

is equivalent to the sequential block

```
do i = 1, N
    do j = 1, M
        call p(i, j)
    end do
end do
```

2.6.2 Parallel execution

A program in the **arb** model can be executed on a shared-memory-model parallel architecture given a language construct that implements general parallel composition as defined in Definition 2.12. In general parallel composition, each element of the composition corresponds to a thread; initiating the composition corresponds to creating a thread for each element and allowing them to execute concurrently, with the composition terminating when all of its component threads have terminated. Language constructs consistent with this form of composition include the **par** and **parfor** constructs of CC++ [21, 19], the `INDEPENDENT` directive of HPF [43], and the `PARALLEL DO` and `PARALLEL SECTIONS` constructs of the Fortran X3H5 proposal [3].

2.6.2.1 Parallel execution using HPF

An **arb**-model program in which all **arb** compositions are of the **arball** form can be transformed into an equivalent program in HPF by replacing `arball` with `forall` and preceding each such block with an `INDEPENDENT` directive, as illustrated in the following examples.

Composition of assignments

The following program block

```
arball (i = 1:N, j = 1:M)
    a(i,j) = i+j
end arball
```

is equivalent to the following HPF program segment

```
!HPF$ INDEPENDENT
forall (i = 1:N, j = 1:M) a(i,j) = i+j
```

Composition of sequential blocks

The following program block

```
arball (i = 1:N, j = 1:M)
    a(i,j) = i+j
    b(i,j) = a(i,j)
end arball
```

is equivalent to the following HPF program segment

```
!HPF$ INDEPENDENT
forall (i = 1:N, j = 1:M)
    a(i,j) = i+j
    b(i,j) = a(i,j)
end forall
```

Here the presence of the `INDEPENDENT` directive means that it is not necessary (as it otherwise would be) to synchronize threads between the statements of the `FORALL` construct.

2.6.2.2 Parallel execution using X3H5 Fortran

An **arb**-model program can be transformed into an equivalent program in the X3H5 notation by replacing `arb` and `end arb` with `PARALLEL SECTIONS`, `SECTION`, and `END PARALLEL SECTIONS` and replacing `arball` and `end arball` with `PARALLEL DO` and `END PARALLEL DO` (nested if necessary), as illustrated in the following examples.

Data-parallel composition of sequential blocks

The following program block

```
arball (i = 1:N)
    a(i) = i
    b(i) = a(i)
end arball
```

is equivalent to the following program segment using the X3H5 extensions to Fortran

```
PARALLEL DO i = 1, N
    a(i) = i
    b(i) = a(i)
END PARALLEL DO
```

Task-parallel composition of sequential blocks

The following program block

```
arb
    seq
        call p1(); call p2()
    end seq
    seq
        call p3(); call p4()
    end seq
end arb
```

is equivalent to the following program segment using the X3H5 extensions to Fortran

```
PARALLEL SECTIONS
SECTION
    call p1()
    call p2()
SECTION
    call p3()
    call p4()
END PARALLEL SECTIONS
```

2.7 Appendix: Program semantics and operational model, details

This section contains a more detailed treatment of the definitions and theorems of Section 2.1.

2.7.1 Notation

We use the following notation:

- One component of our definition of a program is a set of typed variables V. Such a set of variables defines a state space S, in which each state s represents a V-tuple, that is, an assignment of values to variables. For $\{v_1, \ldots, v_N\} \subseteq V$ and $s \in S$, we write

  $$s[v_1/x_1, \ldots, v_N/x_N]$$

  to denote the state formed from s by replacing the value of v_i with x_i, for i such that $1 \leq i \leq N$.

- For $W \subseteq V$ and $s \in S$, we write $s \downarrow W$ to denote the W-tuple formed by projecting s onto W. For $W \subseteq V$, we write $V \setminus W$ to denote the set difference of V and W.

- We use periods to indicate function application, e.g., $f.x$ denotes f applied to x.

- We express quantification as follows:

  "For all" and "there exists":

$$\forall i_1, \ldots, i_N : p.(i_1, \ldots, i_N) : q.(i_1, \ldots, i_N)$$
$$\exists i_1, \ldots, i_N : p.(i_1, \ldots, i_N) : q.(i_1, \ldots, i_N)$$

  denote the intersection and union, respectively, of predicates $q.(i_1, \ldots, i_N)$, where indices $i_1, \ldots, i_N$ range over all values such that $p.(i_1, \ldots, i_N)$.

  Sets and sequences:

$$\{i_1, \ldots, i_N : p.(i_1, \ldots, i_N) : f.(i_1, \ldots, i_N)\}$$

  denotes the set of all $f.(i_1, \ldots, i_N)$, where indices $i_1, \ldots, i_N$ range over all values such that $p.(i_1, \ldots, i_N)$. A similar notation is used for sequences, but using angle brackets ($\langle\ \rangle$) rather than curly braces.

- We employ the following conventions: s (or s_z or s') denotes a program state. P (or P_z or P') denotes a program. C (or C_z or C') denotes a computation of a program. v (or v_z or v') denotes a program variable, with a correspondingly subscripted or primed x denoting its value. q (or q_z or q') denotes a predicate on states.

We sometimes use the proof format of Dijkstra and Scholten [37], which is perhaps most concisely described via an example: Suppose we want to show that a formula A is equal to another formula C by showing that $A = B$ and $B = C$ for some intermediate formula B. We would write this as follows:

$$\begin{array}{ll} & A \\ = & \quad \{ \text{ hint why } A = B \ \} \\ & B \\ = & \quad \{ \text{ hint why } B = C \ \} \\ & C \end{array}$$

2.7.2 Definitions

Definition 2.1′ (Program, revisited).

We define a program P as a 6-tuple $(V, L, InitL, A, PV, PA)$ as in Definition 2.1.

□

Remarks about Definition 2.1′.

- Program action $a = (I_a, O_a, R_a)$ defines a set of state transitions $s \xrightarrow{a} s'$ as follows:

$$(s \xrightarrow{a} s') \equiv ((s \downarrow I_a), (s' \downarrow O_a)) \in R_a \\ \wedge((s' \downarrow (V \setminus O_a)) = (s \downarrow (V \setminus O_a)))$$

- If P is a deterministic program, then for every action a in A, R_a is a partial function from I_a into O_a. The converse is true only if for every state reachable from an initial state (Definition 2.2) at most one action is enabled (Definition 2.3).

- For program action a, I_a includes all program variables that can affect the outcome of a, and O_a includes all program variables whose values can be changed as a result of a. We do not require that I_a and O_a be of minimal size, so it is possible to define two actions corresponding to the same set of state transitions.

- We can also define a program action based on its set of state transitions. Given a set $X \subseteq (S \times S)$ of state transitions, we can define a program action a such that $s \xrightarrow{a} s'$ exactly when $(s, s') \in X$, as follows:

$$\begin{aligned} I_a &= \{v : v \in V \wedge (\exists s, x, x' : to\_states.(s[v/x]) \neq to\_states.(s[v/x'])) : v\} \\ O_a &= \{v : v \in V \wedge (\exists s, s' : (s, s') \in X : (s \downarrow \{v\}) \neq (s' \downarrow \{v\})) : v\} \\ R_a &= \{i, o : (i \text{ is an } I_a - \text{tuple}) \wedge (o \text{ is an } O_a - \text{tuple}) \wedge \\ &\qquad (\forall s, s' \in S : (s \downarrow I_a = i) \wedge (s' \downarrow O_a = o) : (s, s') \in X) \\ &\quad : (i, o)\} \end{aligned}$$

  where

$$to\_states.(s) = \{s' : (s, s') \in X : s'\}$$

  With this approach, I_a and O_a are minimal.

□

Definition 2.2′ (Initial states, revisited).

For program P, we can define the set SI of its initial states thus:

$$SI = \{s : s \downarrow L = InitL : s\}$$

□

Definition 2.3′ (Enabled, revisited).

We write $enabled.(a, s)$ to denote that a is enabled in s, as defined in Definition 2.3.

□

Definition 2.4′ (Computation, revisited).

If $P = (V, L, InitL, A, PV, PA)$, a computation of P is a pair

$$C = (s_0, \langle j : 1 \leq j \leq N : (a_j, s_j)\rangle)$$

in which

- $s_0 \in SI$. We call s_0 the initial state of C and write $init.C = s_0$.
- $\langle j : 1 \leq j \leq N : (a_j, s_j)\rangle$ is a sequence of pairs such that

$$\begin{array}{ll} & \forall j : j \in J : \ a_j \in A \\ \wedge & \forall j : j \in J : s_{j-1} \xrightarrow{a_j} s_j \end{array}$$

  We call these pairs the state transitions of C, and the sequence of actions a_j the actions of C.

  N can be a non-negative integer or ∞. In the former case, we say that C is a finite or terminating computations with length $N+1$ and final state s_N. In the latter case, we say that C is an infinite or nonterminating computation.

  We write $finite.C$ to indicate that C is finite, and for finite C, we write $final.C$ to indicate its final state.

- If C is infinite, the sequence $\langle j : 1 \leq j : (a_j, s_j)\rangle$ satisfies the following fairness requirement:

  If, for some $j \geq 1$ and $a \in A$, *enabled*.(a, s_j), then for some $j' > j$ either $(a, s_{j'})$ is in the above sequence, or $\neg$ *enabled*.$(a, s_{j'})$.

□

Definition 2.5′ (Terminal state, revisited).

We write *terminal*.(s, P) to denote that s is a terminal state of P, as defined in Definition 2.5.

□

Definition 2.6′ (Maximal computation, revisited).

(Same as Definition 2.6.)

□

Definition 2.7′ (Affects, revisited).

We write *affects*.(v, q) to denote that v affects q, and say that *affects*.(v, q) exactly when there exist state s and values x and x' of v such that

$$q.(s[v/x]) \neq q.(s[v/x']) \quad .$$

We write *affects*.(v, E) to denote that v affects E, as defined in Definition 2.7.

□

2.7.3 Specifications and program refinement

Definition 2.8′ (Equivalence of computations, revisited).

For P_1, P_2, and V as described in Definition 2.8, we write $C_1 \overset{V}{\sim} C_2$ to denote that computations C_1 of P_1 and C_2 of P_2 are equivalent with respect to V:

$$((\mathit{init}.C_1 \downarrow V = \mathit{init}.C_2 \downarrow V) \wedge (\neg \mathit{finite}.C_1 \wedge \neg \mathit{finite}.C_2))$$

$$\vee$$

$$((\mathit{init}.C_1 \downarrow V = \mathit{init}.C_2 \downarrow V) \wedge (\mathit{finite}.C_1 \wedge \mathit{finite}.C_2) \wedge (\mathit{final}.C_1 \downarrow V = \mathit{final}.C_2 \downarrow V))$$

□

Remarks about Definition 2.8′.

- Equivalence with respect to a set of variables V is transitive.

□

Theorem 2.9′ (Refinement in terms of equivalent computations, revisited).

If $(V_1 \setminus L_1) \subseteq (V_2 \setminus L_2)$, $P_1 \sqsubseteq P_2$ when for every maximal computation C_2 of P_2 there is a maximal computation C_1 of P_1 such that $C_1 \overset{(V_1 \setminus L_1)}{\sim} C_2$.

□

Proof of Theorem 2.9′.

(See Theorem 2.9.)

□

2.7.4 Program composition

Definition 2.10′ (Composability of programs, revisited).

We say that programs $P_1, \ldots, P_N$, where $P_j = (V_j, L_j, InitL_j, A_j, PV_j, PA_j)$, can be composed exactly when for every $j \neq k$,

$$
\begin{aligned}
& v \in (V_j \cap V_k) \Rightarrow v \text{ has the same type in } V_j \text{ and } V_k \\
\wedge \quad & v \in (V_j \cap V_k) \Rightarrow (v \in PV_j \equiv v \in PV_k) \\
\wedge \quad & L_j \cap L_k = \{\} \\
\wedge \quad & a \in (A_j \cap A_k) \Rightarrow a \text{ is defined in the same way in } A_j \text{ and } A_k
\end{aligned}
$$

□

Remarks about Definition 2.10′.

- If it should be the case that for some $j \neq k$, $L_j \cap L_k \neq \{\}$, we can rename (in P_j or P_k) any variables $v \in L_j \cap L_k$ without changing the meaning of the modified program.

□

2.7.4.1 Sequential composition

Definition 2.11′ (Sequential composition, revisited).

If programs $P_1, \ldots, P_N$, with $P_j = (V_j, L_j, InitL_j, A_j, PV_j, PA_j)$, can be composed (Definition 2.10′), we define their sequential composition $(P_1; \ldots; P_N) = (V, L, InitL, A, PV, PA)$ thus:

- $V = V_1 \cup \ldots \cup V_N \cup L$.
- $L = L_1 \cup \ldots \cup L_N \cup \{En_P, En_1, \ldots, En_N\}$, where $En_P, En_1, \ldots, En_N$ are distinct Boolean variables not otherwise occurring in V.
- $InitL$ is defined by:

$$(\forall j : 1 \leq j \leq N : ((InitL \downarrow L_j = InitL_j) \wedge (InitL \downarrow \{En_j\} = (false)))) \\ \wedge (InitL \downarrow \{En_P\} = (true))$$

- $A = A'_1 \cup \ldots \cup A'_N \cup \{a_{T_0}, \ldots, a_{T_N}\}$, where
  - $A'_j = \{a : a \in A_j : a'\}$, where for $a \in A_j$, $a' = (I_{a'}, O_{a'}, R_{a'})$, with

$$\begin{aligned} I_{a'} &= I_a \cup \{En_j\} \\ O_{a'} &= O_a \\ R_{a'} &= \{i, o : ((i \downarrow I_a, o) \in R_a) \wedge (i \downarrow \{En_j\} = (true)) : (i, o)\} \end{aligned}$$

  - $a_{T_0} = (I_{a_{T_0}}, O_{a_{T_0}}, R_{a_{T_0}})$, with

$$\begin{aligned} I_{a_{T_0}} &= \{En_P\} \\ O_{a_{T_0}} &= \{En_P, En_1\} \\ R_{a_{T_0}} &= \{i, o : (i \downarrow \{En_P\} = (true)) \\ &\qquad \wedge (o \downarrow \{En_P\} = (false)) \wedge (o \downarrow \{En_1\} = (true)) \\ &\quad : (i, o)\} \end{aligned}$$

– We define a_{T_j} in terms of a set of state transitions, as discussed in the remarks following Definition 2.1′.

For a_{T_j} with $1 \leq j \leq N-1$, the required set of state transitions is:

$$\{s : (s \downarrow \{En_j\} = (true)) \wedge terminal.((s \downarrow V_j), P_j) \\ : s \rightarrow s[En_j/false, En_{j+1}/true]\}$$

For a_{T_N}, the required set of state transitions is:

$$\{s : (s \downarrow \{En_N\} = (true)) \wedge terminal.((s \downarrow V_N), P_N) \\ : s \rightarrow s[En_N/false]\}$$

- $PV = PV_1 \cup \ldots \cup PV_N$.
- $PA = \{a : (\exists j :: a \in PA_j) : a'\}$, where a' is as defined above.

□

2.7.4.2 Parallel composition

Definition 2.12′ (Parallel composition, revisited).

If programs $P_1, \ldots, P_N$, with $P_j = (V_j, L_j, InitL_j, A_j, PV_j, PA_j)$, can be composed (Definition 2.10′), we define their parallel composition $(P_1 || \ldots || P_N) = (V, L, InitL, A, PV, PA)$ thus:

- $V = V_1 \cup \ldots \cup V_N \cup L$.
- $L = L_1 \cup \ldots \cup L_N \cup \{En_P, En_1, \ldots, En_N\}$, where $En_P, En_1, \ldots, En_N$ are distinct Boolean variables not otherwise occurring in V.
- $InitL$ is defined by:

$$(\forall j : 1 \leq j \leq N : ((InitL \downarrow L_j = InitL_j) \wedge (InitL \downarrow \{En_j\} = (false)))) \\ \wedge (InitL \downarrow \{En_P\} = (true))$$

- $A = A'_1 \cup \ldots \cup A'_N \cup \{a_{T_0}, \ldots, a_{T_N}\}$, where

– $A'_j = \{a : a \in A_j : a'\}$, where for $a \in A_j$, $a' = (I_{a'}, O_{a'}, R_{a'})$, with

$$\begin{aligned} I_{a'} &= I_a \cup \{En_j\} \\ O_{a'} &= O_a \\ R_{a'} &= \{i, o : ((i \downarrow I_a, o) \in R_a) \wedge (i \downarrow \{En_j\} = (true)) : (i, o)\} \end{aligned}$$

– $a_{T_0} = (I_{a_{T_0}}, O_{a_{T_0}}, R_{a_{T_0}})$, with

$$\begin{aligned} I_{a_{T_0}} &= \{En_P\} \\ O_{a_{T_0}} &= \{En_P, En_1, \ldots, En_N\} \\ R_{a_{T_0}} &= \{i, o : (i \downarrow \{En_P\} = (true)) \wedge (o \downarrow \{En_P\} = (false)) \\ &\qquad \wedge (\forall j : 1 \leq j \leq N : (o \downarrow \{En_j\} = (true))) \\ &\quad : (i, o)\} \end{aligned}$$

– We define a_{T_j} in terms of a set of state transitions, as discussed in the remarks following Definition 2.1′. For a_{T_j}, the required set of state transitions is:

$$\begin{aligned} &\{s : (s \downarrow \{En_j\} = (true)) \wedge terminal.((s \downarrow V_j), P_j) \\ &\qquad : s \rightarrow s[En_j/false]\} \end{aligned}$$

- $PV = PV_1 \cup \ldots \cup PV_N$.
- $PA = \{a : (\exists j :: a \in PA_j) : a'\}$, where a' is as defined above.

□

2.8 arb-compatibility and arb composition, details

This section contains a more detailed treatment of the definitions and theorems of Section 2.2.

2.8.1 Definition of arb-compatibility

Definition 2.13′ (Commutativity of actions, revisited).

Actions a and b of program P are said to *commute* exactly when:

$$\forall s_1, s_2 : s_1 \xrightarrow{b} s_2 : (enabled.(a, s_1) \equiv enabled.(a, s_2))$$

$$\wedge \quad \forall s_1, s_2 : s_1 \xrightarrow{a} s_2 : (\mathit{enabled}.(b, s_1) \equiv \mathit{enabled}.(b, s_2))$$

$$\wedge \quad \forall s_1 : \mathit{enabled}.(a, s_1) \wedge \mathit{enabled}.(b, s_1) :$$

$$((\forall s_2, s_3 : (s_1 \xrightarrow{a} s_2 \wedge s_2 \xrightarrow{b} s_3) : (\exists s_2' :: (s_1 \xrightarrow{b} s_2' \wedge s_2' \xrightarrow{a} s_3)))$$

$$\wedge \ (\forall s_2, s_3 : (s_1 \xrightarrow{b} s_2 \wedge s_2 \xrightarrow{a} s_3) : (\exists s_2' :: (s_1 \xrightarrow{a} s_2' \wedge s_2' \xrightarrow{b} s_3))))$$

□

Definition 2.14′ (arb-compatible, revisited).

Programs $P_1, \ldots, P_N$, where $P_j = (V_j, L_j, \mathit{InitL}_j, A_j, PV_j, PA_j)$, are ***arb**-compatible* exactly when they can be composed (Definition 2.10′) and, for any two actions $a_j \in A_j$ and $a_k \in A_k$ with $j \neq k$, a_j and a_k commute.

□

2.8.2 Equivalence of sequential and parallel composition for arb-compatible components

For the sake of completeness, some material from Section 2.2.2 is repeated here.

Theorem 2.15′ (Parallel $\sim$ sequential for arb-compatible programs, revisited).

If $P_1, \ldots, P_N$ are **arb**-compatible, where $P_j = (V_j, L_j, \mathit{InitL}_j, A_j, PV_j, PA_j)$, then

$$(P_1 || \ldots || P_N) \sim (P_1; \ldots; P_N) \ .$$

□

Proof of Theorem 2.15′.

We write $P_P = (P_1 || \ldots || P_N)$ and $P_S = (P_1; \ldots; P_N)$. From Definition 2.11′ and Definition 2.12′,

$$(V_P = V_S) \wedge (L_P = L_S) \wedge (\mathit{InitL}_P = \mathit{InitL}_S) \wedge (PV_P = PV_S) \wedge (PA_P = PA_S) \ ,$$

so we write $P_P = (V, L, \mathit{InitL}, A_P, PV, PA)$ and $P_S = (V, L, \mathit{InitL}, A_S, PV, PA)$. We proceed as follows:

- We first show (Lemma 2.17′) that for every maximal computation C_S of P_S there is a maximal computation C_P of P_P with $C_S \overset{(V \setminus L)}{\sim} C_P$. From Theorem 2.9′, this establishes that $P_P \sqsubseteq P_S$.
- We then show (Lemma 2.17′) the converse: that for every maximal computation C_S of P_S there is a maximal computation C_S of P_S with $C_S \overset{(V \setminus L)}{\sim} C_S$. From Theorem 2.9′, this establishes that $P_S \sqsubseteq P_S$.
- We then conclude that $P_P \sim P_S$, as desired.

□

We begin by proving two additional lemmas:

Lemma 2.16′ (Reordering of computations, revisited).

(Same as Lemma 2.16.)

□

Lemma 2.28.

For P_P defined as in Theorem 2.15′, if $a_j \in (A'_j \cup \{A_{T_j}\})$ and $a_k \in (A'_k \cup \{A_{T_k}\})$, with $j \neq k$, a_j and a_k commute.

□

Proof of Lemma 2.28.

We define a set of variables $V' = V \setminus \{En_P, En_1, \ldots, En_N\}$, and we consider the various cases:

- $a'_j \in A'_j$ and $a'_k \in A'_k$. Then there are corresponding $a_j \in A_j$ and $a_k \in A_k$, and:

  First we show that if $s_1 \overset{a'_j}{\rightarrow} s_2$, $enabled.(a'_k, s_1) \equiv enabled.(a'_k, s_2)$. From the definition of a'_k,

  $$enabled.(a'_k, s_1) \equiv (enabled.(a_k, s_1 \downarrow V') \wedge (s_1 \downarrow \{En_k\} = (true)) \quad .$$

  Since a_k and a_j commute,

  $$enabled.(a_k, s_1 \downarrow V') \equiv enabled.(a_k, s_2 \downarrow V') \quad .$$

  So since a'_j does not change the value of En_k,

  $$enabled.(a'_k, s_1) \equiv enabled.(a'_k, s_2) \quad .$$

Now we show that

$$enabled.(a'_j, s_1) \wedge enabled(a'_k, s_1) \wedge (s_1 \stackrel{a'_j}{\rightarrow} s_2) \wedge (s_2 \stackrel{a'_k}{\rightarrow} s_3)$$
$$\Rightarrow (\exists s'_2 :: (s_1 \stackrel{a'_k}{\rightarrow} s'_2) \wedge (s'_2 \stackrel{a'_j}{\rightarrow} s_3)) \quad .$$

Clearly

$$enabled.(a_j, s_1 \downarrow V') \wedge enabled(a_k, s_1 \downarrow V')$$
$$\wedge((s_1 \downarrow V') \stackrel{a_j}{\rightarrow} (s_2 \downarrow V')) \wedge ((s_2 \downarrow V') \stackrel{a_k}{\rightarrow} (s_3 \downarrow V')) \quad .$$

Also, $(s_1 \downarrow \{En_j, En_k\}) = (true, true)$, and neither a'_j nor a'_k changes the value of any En_n or En_P.

Since a_j and a_k commute, there is a state s such that

$$((s_1 \downarrow V') \stackrel{a_k}{\rightarrow} (s \downarrow V')) \wedge ((s \downarrow V') \stackrel{a_j}{\rightarrow} (s_3 \downarrow V')) \quad .$$

If we define s'_2 such that

$$((s'_2 \downarrow V') = (s \downarrow V')) \wedge (s'_2 \downarrow (V \setminus V')) = (s_1 \downarrow (V \setminus V')))$$

then s'_2 is the required intermediate stage.

By symmetry, the "vice versa" part of the definition is true: a_k does not enable or disable a_j, and so forth.

- $a'_j \in A'_j$ and $a'_k = a_{T_k}$. Then there is corresponding $a_j \in A_j$, and:

  It is clear from the definitions that if $s_1 \stackrel{a_{T_k}}{\rightarrow} s_2$, $enabled.(a'_j, s_1) \equiv enabled.(a'_j, s_2)$.

  If $s_1 \stackrel{a'_j}{\rightarrow} s_2$, from the definition of a'_j, $(s_1 \downarrow V') \stackrel{a_j}{\rightarrow} (s_2 \downarrow V')$. From the definition of a_{T_k},

  $$enabled.(a_{T_k}, s_1) \equiv ((s_1 \downarrow \{En_k\}) \wedge terminal.((s_1 \downarrow V_k), P_k) \quad .$$

  a_j does not alter the value of En_k, and since a_j commutes with all $a_k \in A_k$,

  $$terminal.((s_1 \downarrow V_k), P_k) \equiv terminal.((s_2 \downarrow V_k), P_k) \quad .$$

  So

  $$enabled.(a_{T_k}, s_1) \equiv enabled.(a_{T_k}, s_2) \quad .$$

Now we show that

$$enabled.(a'_j, s_1) \wedge enabled(a_{T_k}, s_1) \wedge (s_1 \stackrel{a'_j}{\rightarrow} s_2) \wedge (s_2 \stackrel{a_{T_k}}{\rightarrow} s_3) \Rightarrow (\exists s'_2 :: (s_1 \stackrel{a_{T_k}}{\rightarrow} s'_2) \wedge (s'_2 \stackrel{a'_j}{\rightarrow} s_3)) \ .$$

$s'_2 = s_1[En_k/false]$ is the desired intermediate stage.

Finally we show that

$$enabled.(a'_j, s_1) \wedge enabled(a_{T_k}, s_1) \wedge (s_1 \stackrel{a_{T_k}}{\rightarrow} s_2) \wedge (s_2 \stackrel{a'_j}{\rightarrow} s_3)) \Rightarrow (\exists s'_2 :: (s_1 \stackrel{a'_j}{\rightarrow} s'_2) \wedge (s'_2 \stackrel{a_{T_k}}{\rightarrow} s_3)) \ .$$

$s'_2 = s_3[En_k/true]$ is the desired intermediate stage.

- $a'_j = a_{T_j}$ and $a'_k = a_{T_k}$.

  Clearly these two actions commute.

□

Lemma 2.17′ (Sequential refines parallel, revisited).

For P_P and P_S defined as in Theorem 2.15′, if C_S is a maximal computation of P_S, there is a maximal computation C_P of P_P with $C_S \stackrel{(V \setminus L)}{\sim} C_P$.

□

Proof of Lemma 2.17′.

Let C_S be a maximal computation of P_S. We want to produce a computation C_P of P_P such that $C_P \stackrel{(V \setminus L)}{\sim} C_S$. We first observe that the actions of P_S and the actions of P_P have identical names and are in fact identical, with the exception of the actions $\{a_{T_0}, \ldots, a_{T_N}\}$. We distinguish these actions as, for example, $a_{(T_0,P)}$ and $a_{(T_0,S)}$. We construct C_P as follows:

- $init.C_P = init.C_S$. Since $InitL_P = InitL_S$, $init.C_P \in SI_P$.
- The first transition in C_S is $(a_{(T_0,S)}, s_1)$. We define an analogous transition in C_P:

$$(a_{(T_0,P)}, s_1[En_2/true, \ldots, En_N/true])$$

- The rest of C_S is a concatenation of sequences of the following form:

$$\langle n : m_j \leq n \leq M_j : (a_n, s_n) \rangle$$

If C_S is finite, there is one such sequence for each j such that $1 \leq j \leq N$, and every M_j is finite. If C_S is infinite, there is one such sequence for each j such that $1 \leq j \leq N'$, with $N' \leq N$, and M_j is finite for $j < N'$ and infinite for $j = N'$.

Observe that:

$$\begin{aligned} & (s_{(m_j)-1} \downarrow V_j \in SI_j) \\ \wedge \quad & (\forall n : m_j \leq n \leq M_j - 1 : ((s_n \downarrow En_j) = (true))) \\ \wedge \quad & (\forall n : m_j \leq n \leq M_j - 1 : a_n \in A'_j) \\ \wedge \quad & ((M_j < \infty) \Rightarrow (a_{M_j} = a_{(T_j,S)})) \\ \wedge \quad & ((M_j < \infty) \Rightarrow terminal.((s_{(M_j)-1} \downarrow V_j), P_j)) \end{aligned}$$

(If $M_j = \infty$, we interpret $M_j - 1$ as ∞ as well.)

We can map this sequence to a sequence

$$\langle n : m_j \leq n \leq M_j : (a'_n, s'_n) \rangle$$

of transitions of P_P as follows:

$$\begin{aligned} & (\forall n : m_j \leq n \leq M_j : (s'_n = s_n[En_{j+1}/true, \ldots En_N/true]) \\ \wedge \quad & (\forall n : m_j \leq n \leq M_j - 1 : (a'_n = a_n) \\ \wedge \quad & ((M_j < \infty) \Rightarrow (a'_{M_j} = a_{(T_j,P)})) \end{aligned}$$

We observe that:

$$\begin{aligned} & (s'_{(m_j)-1} \downarrow V_j \in SI_j) \\ \wedge \quad & ((M_j < \infty) \Rightarrow terminal.((s'_{(M_j)-1} \downarrow V_j), P_j)) \end{aligned}$$

Observe also that if $M_j < \infty$ and $j < N$,

$$s_{M_j} = s_{m_{(j+1)}-1}$$

so by construction

$$s'_{M_j} = s'_{m_{(j+1)}-1} \ .$$

If C_S is finite, concatenating these sequences $\langle n : m_j \leq n \leq M_j : (a'_n, s'_n)\rangle$ gives us a finite computation C_P of P_P with $\mathit{init}.C_P = \mathit{init}.C_S$. Further,

$$\begin{aligned} & \mathit{final}.C_S = s_{M_N} \\ \wedge \quad & s_{M_N} = s'_{M_N} \\ \wedge \quad & \mathit{terminal}.(s'_{M_N}, P) \ . \end{aligned}$$

So C_P is maximal, and $\mathit{final}.C_P = \mathit{final}.C_S$, the desired result.

If C_S is infinite, concatenating the sequences $\langle n : m_j \leq n \leq M_j : (a'_n, s'_n)\rangle$ almost gives us an infinite computation of P_P, except that this sequence of transitions may violate the fairness requirement that is part of our definition of computation: Since $M_{N'} = \infty$, for $j > N'$, no action from A_j can ever execute, even if one is enabled. We can, however, produce from our concatenation of sequences a (fair) computation of P_P: We observe that each sequence

$$\langle n : m_j \leq n \leq M_j : (a'_n, s'_n)\rangle$$

corresponds to a computation of P_j with initial state $\mathit{init}.C_S \downarrow V_j$. For each $j > N'$, define a similar sequence corresponding to some (arbitrarily-chosen) maximal computation of P_j with initial state $\mathit{init}.C_S \downarrow V_j$, followed (if finite) by a transition corresponding to A_{T_j}. Each sequence

$$\langle n : m_j \leq n \leq M_j : (a'_n, s'_n)\rangle$$

corresponds to a sequence of actions in $(A'_j \cup \{A_{T_j}\})$. We now have, for each j, a sequence of actions

$$a_{(j,1)}, a_{(j,2)}, \cdots \ ,$$

where each $a_{(j,n)}$ is in $(A'_j \cup \{A_{T_j}\})$. Consider the sequence α of actions produced by alternating elements from these sequences:

$$\alpha = a_{(1,1)}, a_{(2,1)}, \ldots, a_{(N,1)}, a_{(1,2)}, a_{(2,2)}, \ldots, a_{(N,2)}, \ldots$$

(Observe that if $n > M_j$, there is no $a_{(j,n)}$; in this case, we simply continue with the next element in the above sequence.) We observe that, from Lemma 2.28, actions in $(A'_j \cup \{A_{T_j}\})$ commute with actions in $(A'_k \cup \{A_{T_k}\})$, for $j \neq k$. In particular, actions in $(A'_j \cup \{A_{T_j}\})$ neither enable nor disable actions in $(A'_k \cup \{A_{T_k}\})$. Thus, we can generate from α a sequence

of transitions of P_P from initial state $init.C_S$:

- Let $s_0 = init.C_S$, and
- for $n \geq 1$, choose some s_n such that $s_{n-1} \xrightarrow{b_i} s_n$, where b_i is the i-th action in α.

Since this sequence of transitions clearly satisfies the fairness requirement, it together with its initial state forms an infinite computation C_P of P_P with the same initial state as C_S, the desired result.

So we have produced a maximal computation C_P of P_P such that $C_P \overset{(V \setminus L)}{\sim} C_S$.

□

Lemma 2.18′ (Parallel refines sequential, revisited).

For P_S and P_P defined as in Theorem 2.15′, if C_P is a maximal computation of P_P, there is a maximal computation C_S of P_S with $C_P \overset{(V \setminus L)}{\sim} C_S$.

□

Proof of Lemma 2.18′.

We first construct from C_P an equivalent sequence of transitions of P_P with the property that for any pair of transitions

$$((a_j, s_{n_j}), (a_k, s_{n_k}))$$

(not necessarily consecutive) of C_P, where

$$(a_j \in (A'_j \cup \{A_{T_j}\}) \wedge (a_k \in (A'_k \cup \{A_{T_k}\})$$

we have

$$(n_j < n_k) \Rightarrow (j \leq k) \quad .$$

We will refer to such a pair of transitions as being *in order*. We will then map this sequence of transitions to a computation of C_S.

Constructing the sequence of transitions. We construct the sequence of transitions in a manner analogous to a well-known nondeterministic sorting algorithm (as discussed in, e.g., [22]) in which an array is sorted by repeatedly choosing one out-of-sequence pair and exchanging its elements.

We first consider finite computations. Suppose C_P is a finite computation of P. Let $M.(C_P)$ be the number of pairs of state transitions of C_P that are not in order. For m such that $1 \leq m \leq M.(C_P)$, define $C_P^{(m)}$ thus:

- $C_P^{(0)} = C_P$.
- For m such that $1 \leq m \leq M.(C_P)$, if $M.(C_P^{(m-1)}) = 0$, define $C_P^{(m)} = C_P^{(m-1)}$.

  If $M.(C_P^{(m-1)}) > 0$, there is at least one pair of consecutive state transitions

  $$((a_j, s_n), (a_k, s_{n+1}))$$

  in $C_P^{(m-1)}$ such that

  $$(a_j \in (A'_j \cup \{A_{T_j}\}) \wedge (a_k \in (A'_k \cup \{A_{T_k}\}) \wedge (j > k) \ .$$

  From Lemma 2.28, a_j and a_k commute, and from Lemma 2.16′, we can construct $C_P^{(m)}$ with the same initial state as $C_P^{(m-1)}$ and the same sequence of state transitions, except that we replace the pair

  $$((a_j, s_n), (a_k, s_{n+1}))$$

  with the pair

  $$((a_k, s'_n), (a_j, s_{n+1})) \ .$$

We observe that for m such that $1 \leq m \leq M.(C_P)$,

$$C_P^{(m)} \overset{(V \setminus L)}{\sim} C_P^{(m-1)}$$

since the two computations have the same final state, and from the transitivity of this equivalence relation,

$$C_P^{(m)} \overset{(V \setminus L)}{\sim} C_P \ .$$

Further, either $M.(C_P^{(m)}) = 0$, or $M.(C_P^{(m)}) \leq M.(C_P^{(m-1)}) - 1$, so $M.(C_P^{(M)}) = 0$. We have thus produced a computation $C_P^{(M)}$ of P_P such that $C_P^{(M)} \overset{(V \setminus L)}{\sim} C_P$ and every pair of transitions in $C_P^{(M)}$ is in order. Observe also that since C_P is maximal, so is $C_P^{(M)}$.

Now consider the case of infinite C_P. Because of fairness considerations, we may not be able to produce from C_P an infinite computation C'_P of P_P with the property that every pair of transitions in C'_P is in order. However, we can produce a sequence of transitions of P_P from the initial state of C_P with this property, which is all we need. We proceed as follows: Since C_P is infinite, there is at least one m such that $1 \leq m \leq N$ and C_P contains infinitely many actions from A'_m. Choose the

smallest such m. For j such that $1 \leq j \leq m-1$, C_P contains a finite sequence

$$a_{(j,1)}, a_{(j,2)}, \ldots, a_{(j,n_j)}$$

of actions from A'_j. C_P also contains an infinite sequence

$$a_{(m,1)}, a_{(m,2)}, \ldots$$

of actions from A'_m. We can concatenate these sequences into a single infinite sequence

$$\alpha = a_{(1,1)}, \ldots, a_{(1,n_1)}, a_{(2,1)}, \ldots, a_{(2,n_2)}, \ldots a_{(m-1,1)}, \ldots, a_{(m-1,n_{m-1})}, a_{(m,1)}, a_{(m,2)}, \ldots$$

Because a_j and a_k commute whenever $a_j \in (A'_j \cup \{a_{T_j}\})$ and $a_k \in (A'_k \cup \{a_{T_k}\})$ and $j \neq k$, (and thus actions from $(A'_j \cup \{a_{T_j}\})$ do not change the enabled status of actions from $(A'_k \cup \{a_{T_k}\})$ where $j \neq k$), we can define a sequence τ of transitions of P_P thus:

- Let $s_0 = \mathit{init}.C_P$, and
- for $n \geq 1$, choose some s_n such that $s_{n-1} \xrightarrow{b_i} s_n$, where b_i is the i-th action in α.

This gives us an infinite sequence τ of transitions of P_P with the desired property.

Observe that for j such that $1 \leq j \leq m-1$, the sequence $a_{(j,1)}, a_{(j,2)}, \ldots, a_{(j,n_j)}$ corresponds to a maximal terminating computation of P_j: Given that $(a_{(j,n_j)}, s)$ appears in C_P, we must have $\mathit{terminal}.((s \downarrow V_j), P_j)$, since if any action from A_j is enabled in $s \downarrow V_j$, the corresponding action of A'_j is enabled in s, and by the fairness requirement and the **arb**-compatibility restrictions must eventually appear in C_P, contrary to hypothesis. Further, we observe that for the corresponding transition $(a_{(j,n_j)}, s')$ in τ, we must have $\mathit{terminal}.((s' \downarrow V_j), P_j)$, again from the **arb**-compatibility restriction that actions from one component do not enable or disable actions from another component.

Mapping the sequence to a computation of P_S. Having constructed a sequence τ of transitions of P_P starting from $\mathit{init}.C_P$ with the property that every pair of transitions in the sequence is in order, we can now construct a computation C_S of P_S, with $C_S \overset{(V \setminus L)}{\sim} C_P$, as follows. (The proof of this claim is very similar to the proof of Lemma 2.17′.)

We first observe that the actions of P_P and the actions of P_S have identical names and are in fact identical, with the exception of the actions $\{a_{T_0}, \ldots, a_{T_N}\}$. We distinguish these actions as, for example, $a_{(T_0,P)}$ and $a_{(T_0,S)}$.

- $\mathit{init}.C_S = \mathit{init}.C_P$. Since $\mathit{InitL}_S = \mathit{InitL}_P$, $\mathit{init}.C_S \in SI_S$.

- The first transition in τ is $(a_{(T_0,P)}, s_1)$. We define an analogous transition in C_S:

$$(a_{(T_0,S)}, s_1[En_2/false, \ldots, En_N/false])$$

- The rest of τ is a concatenation of sequences of the following form:

$$\langle n : m_j \leq n \leq M_j : (a_n, s_n) \rangle$$

If C_P is finite, there is one such sequence for each j such that $1 \leq j \leq N$, and every M_j is finite. If C_P is infinite, there is one such sequence for each j such that $1 \leq j \leq N'$, with $N' \leq N$, and M_j is finite for $j < N'$ and infinite for $j = N'$.

Observe that:

$$\begin{aligned}
& (s_{(m_j)-1} \downarrow V_j \in SI_j) \\
\wedge \quad & (\forall n : m_j \leq n \leq M_j - 1 : (\forall k : j \leq k \leq N : (s_n \downarrow En_k) = (true))) \\
\wedge \quad & (\forall n : m_j \leq n \leq M_j - 1 : a_n \in A'_j) \\
\wedge \quad & ((M_j < \infty) \Rightarrow (a_{M_j} = a_{(T_j,P)})) \\
\wedge \quad & ((M_j < \infty) \Rightarrow terminal.((s_{(M_j)-1} \downarrow V_j), P_j))
\end{aligned}$$

(If $M_j = \infty$, we interpret $M_j - 1$ as ∞ as well.)

The truth of the last conjunct $(terminal.((s_{(M_j)-1} \downarrow V_j), P_j))$ is less obvious than is the case for the analogous conjunct in the proof of Lemma 2.17′, but we reason as follows: If $\neg terminal.((s_{(M_j)-1} \downarrow V_j), P_j)$, then in state $s_{(M_j)-1}$ there is an enabled action from A'_j. If τ is finite, τ corresponds to a maximal computation of P_P (by construction above), and so this action from A'_j must occur later in τ, since no action by another A'_k can disable it (by **arb**-compatibility). This is impossible, since all pairs of transitions of τ are in order. If τ is infinite but $M_j < \infty$, by construction of τ we have the desired result, since j must be such that C_P contains only finitely many actions from A'_j, the last of which produces a terminal state of P_j, as discussed earlier.

Continuing, we can map this sequence to a sequence

$$\langle n : m_j \leq n \leq M_j : (a'_n, s'_n) \rangle$$

of transitions of P_S as follows:

$$\begin{aligned} & (\forall n : m_j \leq n \leq M_j : (s'_n = s_n[En_{j+1}/false, \ldots En_N/false]) \\ \wedge \quad & (\forall n : m_j \leq n \leq M_j - 1 : (a'_n = a_n)) \\ \wedge \quad & ((M_j < \infty) \Rightarrow (a_{M_j} = a_{(T_j,S)})) \end{aligned}$$

We observe that:

$$\begin{aligned} & (s'_{(m_j)-1} \downarrow V_j \in SI_j) \\ \wedge \quad & ((M_j < \infty) \Rightarrow terminal.((s'_{(M_j)-1} \downarrow V_j), P_j)) \end{aligned}$$

Observe also that if $M_j < \infty$ and $j < N$,

$$s_{M_j} = s_{m_{(j+1)}-1}$$

so by construction

$$s'_{M_j} = s'_{m_{(j+1)}-1} \ .$$

So concatenating these sequences $\langle n : m_j \leq n \leq M_j : (a'_n, s'_n) \rangle$ gives us a sequence of transitions of P_S with $init.C_S = init.C_P$.

If C_P is finite, so is this sequence of transitions, so it forms a finite computation C_P, and:

$$\begin{aligned} & final.C_P = s_{M_N} \\ \wedge \quad & s_{M_N} = s'_{M_N} \\ \wedge \quad & terminal.(s'_{M_N}, P_S) \end{aligned}$$

So C_S is maximal, and $final.C_S = final.C'_P$.

If C_P is infinite, so is this sequence of transitions, and it (unlike τ) meets the fairness requirement, since in P_S actions from P_j become enabled only after P_{j-1} terminates. So this sequence of transitions forms an infinite computation of P_S.

So we have produced a computation C_S of P_S such that $C_S \overset{(V \setminus L)}{\sim} C'_P$.

□

2.8.3 Simpler sufficient conditions for arb-compatibility

Definition 2.22′ (Variables read by P, revisited).

For program $P = (V, L, InitL, A, PV, PA)$, we define the set of variables read by P thus:

$$VR = \cup_{a \in A} I_a$$

□

Definition 2.23′ (Variables written by P, revisited).

For program $P = (V, L, InitL, A, PV, PA)$, we define the set of variables written by P thus:

$$VW = \cup_{a \in A} O_a$$

□

Definition 2.24′ (Programs that share only read-only variables, revisited).

If programs $P_1, \ldots, P_N$, where $P_j = (V_j, L_j, InitL_j, A_j, PV_j, PA_j)$, can be composed (Definition 2.10′), and we have that

$$\forall j, k : j \neq k : (VW_j \cap (VR_k \cup VW_k) = \{\}),$$

then we say that $P_1, \ldots, P_N$ *share only read-only variables.*

□

Theorem 2.25′ (arb-compatibility and shared variables, revisited).

If programs $P_1, \ldots, P_N$ share only read-only variables (Definition 2.24′), then $P_1, \ldots, P_N$ are **arb**-compatible.

□

Proof of Theorem 2.25′.

Given programs $P_1, \ldots, P_N$ that satisfy the condition, we want to show that for any $a_j \in A_j$ and $a_k \in A_k$ with $j \neq k$, a_j and a_k commute.

First we consider whether a_j can affect the enabled status of a_k. If $s_1 \stackrel{a_j}{\rightarrow} s_2$, from the restrictions on shared variables,

$$(s_2 \downarrow I_{a_k}) = (s_1 \downarrow I_{a_k}) \ .$$

Since

$$\mathit{affects}.(v, \mathit{enabled}.(a_k, s)) \Rightarrow (v \in I_{a_k})$$

clearly

$$\mathit{enabled}.(a_k, s_1) \equiv \mathit{enabled}.(a_k, s_2) \ .$$

By symmetry, a_k cannot affect the enabled status of a_j.

Now consider the situation in which we have

$$\mathit{enabled}.(a_j, s_1) \wedge \mathit{enabled}.(a_k, s_1) \wedge (s_1 \stackrel{a_j}{\rightarrow} s_2) \wedge (s_2 \stackrel{a_k}{\rightarrow} s_3) \ .$$

We want state s_2' such that

$$(s_1 \stackrel{a_k}{\rightarrow} s_2') \wedge (s_2' \stackrel{a_j}{\rightarrow} s_3) \ .$$

Define s_2' thus:

$$\begin{aligned} & \forall v : v \in O_{a_k} : (s_2' \downarrow \{v\}) = (s_3 \downarrow \{v\}) \\ \wedge \quad & \forall v : v \notin O_{a_k} : (s_2' \downarrow \{v\}) = (s_1 \downarrow \{v\}) \end{aligned}$$

$s_1 \stackrel{a_k}{\rightarrow} s_2'$ exactly when

$$(((s_1 \downarrow I_{a_k}), (s_2' \downarrow O_{a_k})) \in R_{a_k}) \tag{2.1}$$

$$\wedge \quad ((s_2' \downarrow (V \setminus O_{a_k})) = (s_1 \downarrow (V \setminus O_{a_k}))) \tag{2.2}$$

(2.2) holds by construction of s_2'. (2.1) holds because:

$$\begin{aligned} & ((s_1 \downarrow I_{a_k}) = (s_2 \downarrow I_{a_k})) \\ \wedge \quad & ((s_2' \downarrow O_{a_k}) = (s_3 \downarrow O_{a,k})) \\ \wedge \quad & (((s_2 \downarrow I_{a_k}), (s_3 \downarrow O_{a,k})) \in R_{a_k}) \end{aligned}$$

$s_2' \xrightarrow{a_j} s_3$ exactly when

$$(((s_2' \downarrow I_{a_j}), (s_3 \downarrow O_{a_j})) \in R_{a_j}) \tag{2.3}$$

$$\wedge \quad ((s_3 \downarrow (V \setminus O_{a_j})) = (s_2' \downarrow (V \setminus O_{a_j}))) \tag{2.4}$$

(2.3) holds because:

$$\begin{aligned} & ((s_2' \downarrow I_{a_j}) = (s_1 \downarrow I_{a_j})) \\ \wedge \quad & ((s_3 \downarrow O_{a_j}) = (s_2 \downarrow O_{a_j})) \\ \wedge \quad & (((s_1 \downarrow I_{a_j}), (s_2 \downarrow O_{a_j})) \in R_{a_j})) \end{aligned}$$

(2.4) holds by construction of s_2' and because

$$\forall v : (v \notin O_{a_k}) \wedge (v \notin O_{a_j}) : ((s_3 \downarrow \{v\}) = (s_1 \downarrow \{v\}))$$

By symmetry, a similar construction applies to computations in which a_k is performed first.

□

2.9 Appendix: Dijkstra's guarded-command language and our model, details

In this section we sketch definitions in our model for some of the commands and constructors of Dijkstra's guarded-command language [35, 37].

2.9.1 Simple commands

Definition 2.29 (Skip).

We define program $skip = (V, L, InitL, A, PV, PA)$ as follows:

- $V = L$.
- $L = \{En_{skip}\}$, where En_{skip} is a Boolean variable.
- $InitL = (true)$.
- $A = \{a\}$, where

$$I_a \quad = \quad \{En_{skip}\}$$

$$\begin{aligned} O_a &= \{En_{skip}\} \\ R_a &= \{((true),(false))\} \end{aligned}$$

- $PV = \{\}$.
- $PA = \{\}$.

□

Definition 2.30 (Assignment).

We define program $P = (V, L, InitL, A, PV, PA)$ for $(y := E)$ as follows:

- $V = \{v_1, \ldots, v_N\} \cup \{y\} \cup L$, where $\{v_1, \ldots, v_N\} = \{v : affects.(v, E) : v\}$.
- $L = \{En_P\}$, where En_P is a Boolean variable not otherwise occurring in V.
- $InitL = (true)$.
- $A = \{a\}$, where

$$\begin{aligned} I_a &= \{En_P\} \cup \{v_1, \ldots, v_N\} \\ O_a &= \{En_P, y\} \\ R_a &= \{x_1, \ldots, x_N :: ((true, x_1, \ldots, x_N), (false, E.(x_1, \ldots, x_N))\} \end{aligned}$$

  and $x_1, \ldots, x_N$ is an assignment of values to the variables in $v_1, \ldots, v_N$.

- $PV = \{\}$.
- $PA = \{\}$.

□

Definition 2.31 (Abort).

We define program $abort = (V, L, InitL, A, PV, PA)$ as follows:

- $V = L$
- $L = \{En_{abort}\}$, where En_{abort} is a Boolean variable.
- $InitL = (true)$.

- $A = \{a\}$, where

$$\begin{aligned} I_a &= \{En_{abort}\} \\ O_a &= \{\} \\ R_a &= \{(true), ()\} \end{aligned}$$

- $PV = \{\}$.
- $PA = \{\}$.

□

2.9.2 Alternative composition (IF)

First we need an additional preliminary definition:

Definition 2.32 (Composability of guards with programs).

We say that guards $b_1, \ldots, b_N$, where b_j is a Boolean expression with variables W_j, can be composed with programs $P_1, \ldots, P_N$, where $P_j = (V_j, L_j, InitL_j, A_j, PV_j, PA_j)$, exactly when $P_1, \ldots, P_N$ can be composed (Definition 2.10′) and for all j:

$$v \in W_j \Rightarrow (\exists k :: v \in V_k \wedge v \text{ has the same type in } W_j \text{ and } V_k) \ .$$

□

Definition 2.33 (Alternative composition).

Our definition of alternative composition is analogous to the definition of sequential composition in Definition 2.11′. Given programs $P_1, \ldots, P_N$, with $P_j = (V_j, L_j, InitL_j, A_j, PV_j, PA_j)$, and Boolean expressions $b_1, \ldots, b_N$ such that $P_1, \ldots, P_N$ can be composed (Definition 2.10′) and $b_1, \ldots, b_N$ can be composed with $P_1, \ldots, P_N$ (Definition 2.32), we define program $P = (V, L, InitL, A, PV, PA)$ for

$$\textbf{if } []_j \ b_j \ \rightarrow \ P_j \textbf{ fi}$$

as follows:

- $V = V_1 \cup \ldots \cup V_N \cup L$.

- $L = L_1 \cup \ldots \cup L_N \cup \{En_P, En_{abort}, En_1, \ldots, En_N\}$, where $En_P, En_{abort}, En_1, \ldots, En_N$ are distinct Boolean variables not otherwise occurring in V.

- $InitL$ is defined by:

$$\begin{aligned} & (\forall j : 1 \leq j \leq N : ((InitL \downarrow L_j = InitL_j) \wedge (InitL \downarrow \{En_j\} = (false)))) \\ \wedge \quad & (InitL \downarrow \{En_P, En_{abort}\} = (true, false)) \end{aligned}$$

- A consists of the following actions:

  - An action a_{abort} for the case in which initially none of the guards is $true$. $s \stackrel{a_{abort}}{\rightarrow} s'$ exactly when

$$\begin{aligned} & ((s \downarrow \{En_P\} = (true)) \wedge (\forall j :: \neg b_j.s) \wedge (s' = s[En_P/false, En_{abort}/true])) \\ & \vee \\ & ((s \downarrow \{En_{abort}\} = (true)) \wedge (s' = s)) \quad . \end{aligned}$$

  - For each j such that $1 \leq j \leq N$, an action a_{start_j} for the case in which guard b_j is initially $true$. $s \stackrel{a_{start_j}}{\rightarrow} s'$ exactly when

$$(s \downarrow \{En_P\} = (true)) \wedge b_j.s \wedge (s \downarrow V_j \in SI_j) \wedge (s' = s[En_P/false, En_j/true]) \quad .$$

  - For each j such that $1 \leq j \leq N$, an action a_{end_j} that terminates the IF composition after the selected P_j (started by an a_{start_j} action) completes. $s \stackrel{a_{end_j}}{\rightarrow} s'$ exactly when

$$(s \downarrow \{En_j\} = (true)) \wedge terminal.((s \downarrow V_j), P_j) \wedge (s' = s[En_j/false]) \quad .$$

  - For each action a_j in A_j, a corresponding action a'_j, defined as for sequential composition: $s \stackrel{a'_j}{\rightarrow} s'$ exactly when

$$\begin{aligned} & (s \downarrow \{En_j\} = (true)) \\ & \wedge((s \downarrow V_j) \stackrel{a_j}{\rightarrow} (s' \downarrow V_j)) \wedge ((s' \downarrow (V \setminus V_j)) = (s \downarrow (V \setminus V_j))) \quad . \end{aligned}$$

- $PV = PV_1 \cup \ldots \cup PV_N$.

- PA contains exactly those actions a' derived (as described above) from the actions a of $PA_1 \cup \ldots \cup PA_N$.

□

2.9.3 Repetition (DO)

Definition 2.34 (Repetition).

Our definition of repetition is analogous to the definition of sequential composition in Definition 2.11′ and the definition of alternative composition in Definition 2.33. Given program $P_{body} = (V_{body}, L_{body}, InitL_{body}, A_{body})$ and Boolean expression b such that b can be composed with P_{body} (Definition 2.32), we define program $P = (V, L, InitL, A, PV, PA)$ for

$$\textbf{do } b \ \rightarrow \ P \textbf{ od}$$

as follows:

- $V = V_{body} \cup L$.
- $L = L_{body} \cup \{En_P, En_{body}\}$, where En_P, En_{body} are distinct Boolean variables not otherwise occurring in V.
- $InitL$ is defined by:

$$(InitL \downarrow L_{body} = InitL_{body}) \wedge (InitL \downarrow \{En_P, En_{body}\} = (true, false))$$

- A consists of the following actions:
  - An action a_{exit} to exit the loop. $s \stackrel{a_{exit}}{\rightarrow} s'$ exactly when

$$(s \downarrow \{En_P\} = (true)) \wedge \neg b.s \wedge (s' = s[En_P/false]) \quad .$$

  - An action a_{start} to start a loop iteration. $s \stackrel{a_{start}}{\rightarrow} s'$ exactly when

$$(s \downarrow \{En_P\} = (true)) \wedge b.s \wedge (s' = s[En_P/false, En_{body}/true]) \ .$$

  - An action a_{cycle} to return to the beginning of the loop and test the guard again. $s \stackrel{a_{cycle}}{\rightarrow} s'$ exactly when

$$\begin{aligned}&(s \downarrow \{En_{body}\} = (true)) \wedge terminal.((S \downarrow V_{body}), P_{body})\\ &\wedge (s' = s[En_{body}/false, En_P/true, L_{body}/InitL_{body}]) \quad .\end{aligned}$$

    (In the equality for s', "$L_{body}/InitL_{body}$" indicates that the values of all variables in L_{body} are to be replaced by their values in $InitL_{body}$.)

– For each action a_{body} in A_{body}, a corresponding action a'_{body}, defined as for sequential composition: $s \overset{a'_{body}}{\rightarrow} s'$ exactly when

$$(s \downarrow \{En_{body}\} = (true))$$
$$\wedge((s \downarrow V_{body}) \overset{a_{body}}{\rightarrow} (s' \downarrow V_{body})) \wedge ((s' \downarrow (V \setminus V_{body})) = (s \downarrow (V \setminus V_{body}))) \quad .$$

- $PV = PV_{body}$.
- PA contains exactly those actions a' derived (as described above) from the actions a of PA_{body}.

□

Chapter 3

A collection of useful transformations

In the preceding chapter we described the first step of our programming methodology: expressing the desired computation in what we call the **arb** model (sequential constructs plus **arb** composition). Because **arb** composition can be implemented as either sequential or parallel composition (with equivalent results), programs in the **arb** model can be executed as parallel programs. However, they may not make effective use of typical parallel architectures, so our methodology also addresses the question of how to transform them into programs that make better use of parallel architectures. Chapter 4 and Chapter 5 describe the eventual goal of such transformations — programs suitable for execution on a shared-memory-model architecture with barrier synchronization (the **par** model of Chapter 4) or a distributed-memory-model architecture with message-passing (the subset **par** model of Chapter 5). This chapter presents a collection of transformations useful in the step-by-step conversion of an initial **arb**-model program into a program in one of these models. These transformations have the following useful characteristics:

- They can be viewed as semantics-preserving transformations for sequential programs, so arguments for their correctness can be given based on the techniques of sequential stepwise refinement.
- They produce programs that can be executed sequentially, so their results can be verified and debugged using sequential tools and techniques.

For each transformation or class of transformations presented in this chapter, we present:

- The transformation.
- A discussion of its utility.

- An argument for its correctness (i.e., an argument that it produces a program that refines the original program).
- An example or examples of its use.

We also sketch without proof some additional transformations. This collection is not intended as an exhaustive list of all possible useful transformations, but rather as a representative collection that is also sufficient to address a range of typical application programs.

It is important also to note the role of archetypes in the transformation process: An archetype can provide, for the class of programs whose common features it abstracts, not only a pattern for the **arb**-model program that is the first step in our program-development methodology, but also a strategy for selecting and applying appropriate transformations and a pattern for the eventual shared-memory or distributed-memory program. The archetype can then guide the transformation process toward a result that is known to be efficient, with the process (applying a sequence of semantics-preserving transformations) guaranteeing the correctness of the result.

3.1 Removal of superfluous synchronization

3.1.1 Motivation

If there is significant cost associated with executing a parallel composition (because of thread creation), then program efficiency can clearly be improved by combining a sequence of **arb** compositions of N elements into a single **arb** composition of N elements, as shown in the following theorem, when it is possible to do so without changing the meaning of the program.

3.1.2 Definition and argument for correctness

Theorem 3.1 (Removal of superfluous synchronization).

If $P_1, \ldots, P_N$ are **arb**-compatible, and $Q_1, \ldots, Q_N$ are **arb**-compatible, and the sequential compositions $\mathbf{seq}(P_1, Q_1)$, $\ldots$, $\mathbf{seq}(P_N, Q_N)$ are **arb**-compatible, then

$$\mathbf{seq}(\mathbf{arb}(P_1, \ldots, P_N), \mathbf{arb}(Q_1, \ldots, Q_N))$$

$$\sim$$

$$\mathbf{arb}(\mathbf{seq}(P_1, Q_1), \ldots, \mathbf{seq}(P_N, Q_N))$$

□

Proof of Theorem 3.1.

First we observe that for $j \neq k$, $\mathbf{seq}(P_j, Q_j)$ and $\mathbf{seq}(P_k, Q_k)$ are **arb**-compatible (from the hypothesis and the definition of **arb**-compatibility, Definition 2.14), and hence P_j and Q_k are **arb**-compatible (from the definitions of sequential composition and **arb**-compatibility, Definition 2.11 and Definition 2.14). From the definition and commutativity of **arb** composition (Theorem 2.15 and Theorem 2.20), then, $(P_j; Q_k) \sim (Q_k; P_j)$. We can then calculate thus:

$\mathbf{seq}(\mathbf{arb}(P_1, \ldots, P_N), \mathbf{arb}(Q_1, \ldots, Q_N))$

$\sim$ { Theorem 2.15 and associativity of sequential composition }

$P_1; \ldots; P_N; Q_1; \ldots; Q_N$

$\sim$ { as noted above }

$P_1; \ldots; P_{N-1}; Q_1; P_N; Q_2; \ldots; Q_N$

$\sim$ { repeating the above step repeatedly }

$P_1; Q_1; \ldots; P_N; Q_N$

$\sim$ { associativity of sequential composition, Theorem 2.15, and hypothesis }

$\mathbf{arb}(\mathbf{seq}(P_1, Q_1), \mathbf{seq}(P_N, Q_N))$

□

3.1.3 Example

Let program P be the following program:

```
integer a(N), b(N), c(N)
arball (i = 1 : N)
    b(i) = a(i)
end arball
arball (i = 1 : N)
    c(i) = b(i)
end arball
```

Then P is equivalent to the following program P':

```
integer a(N), b(N), c(N)
arball (i = 1 : N)
    b(i) = a(i)
    c(i) = b(i)
end arball
```

3.2 Change of granularity

3.2.1 Motivation

If (1) the number of elements in an **arb** composition is large compared to the number of processors available for execution, and (2) the cost of creating a separate thread for each element of the composition is relatively high, then we can improve the efficiency of the program by reducing the number of threads required, that is, by changing the granularity of the program.

3.2.2 Definition and argument for correctness

We can change the granularity of an **arb**-model program by transforming an **arb** composition of N elements into a combination of **arb** composition (of fewer than N elements) and sequential composition, as described in the following theorem.

Theorem 3.2 (Change of granularity).

If $P_1, \ldots, P_N$ are **arb**-compatible, and we have integers $j_1, j_2, \ldots, j_M$ such that $(1 < j_1) \wedge (j_1 < j_2) \wedge \ldots \wedge (j_M < N)$, then

$$
\begin{array}{l}
\quad \mathbf{arb}(P_1, \ldots, P_N) \\
\sim \\
\quad \mathbf{arb}(\\
\qquad \mathbf{seq}(P_1, \ldots, P_{j_1}), \\
\qquad \mathbf{seq}(P_{j_1+1}, \ldots, P_{j_2}), \\
\qquad \ldots, \\
\qquad \mathbf{seq}(P_{j_M+1}, \ldots, P_N) \\
\quad)
\end{array}
$$

□

Proof of Theorem 3.2.

This follows immediately from the associativity of **arb** composition (Theorem 2.19) and the equivalence of sequential and **arb** composition (Theorem 2.15).

□

3.2.3 Example

Continuing the example of Section 3.1, let program P be the following program:

```
integer a(N), b(N), c(N)
arball (i = 1 : N)
    b(i) = a(i)
    c(i) = b(i)
end arball
```

Then P is equivalent to the following program P':

```
integer a(N), b(N), c(N)
arb
    do i = 1, N/2
        b(i) = a(i)
        c(i) = b(i)
    end do
    do i = N/2 + 1, N
        b(i) = a(i)
        c(i) = b(i)
    end do
end arb
```

If only two processors are available, program P' is likely to be more efficient than P, since P implies the creation of N threads, while P' implies the creation of only 2 threads.

3.3 Data distribution and duplication

3.3.1 Motivation

In order to transform a program in the **arb** model into a program suitable for execution on a distributed-memory architecture, we must partition its variables into distinct groups, each corresponding to an address space (and hence to a process). Such partitioning is essential to producing a program suitable for execution on a distributed-memory architecture, but it may also improve efficiency on some shared-memory architectures, for example those in which each processor has a separate cache, since programs with a high degree of data locality may make more effective use of such caches. Chapter 5 describes the characteristics such a partitioning should have in order to permit execution on a distributed-memory architecture; in this chapter we discuss only the mechanics

of the partitioning, that is, transformations that effect partitioning while preserving program correctness. These transformations fall into two categories: data distribution, in which variables of the original program are mapped one-to-one onto variables of the transformed program; and data duplication, in which the map is one-to-many, that is, in which some variables of the original program are duplicated in the transformed program.

3.3.2 Data distribution: definition and argument for correctness

The transformations required to effect data distribution are in essence renamings of program variables, in which variables of the original program are mapped one-to-one to variables of the transformed program. The most typical use of data distribution is in partitioning non-atomic data objects such as arrays: Each array is divided into *local sections*, one for each process, and a one-to-one map is defined between the elements of the original array and the elements of the (disjoint) union of the local sections. Figure 3.1 shows an example of such partitioning. The shaded element illustrates the one-to-one map between the original array and its partitioning: It is mapped from position (3,6) of the original array to position (1,2) in array section (2,2). That such a renaming operation does not change the meaning of the program is clear, although if elements of the array are referenced via index variables, some care must be taken to ensure that they (the index variables) are transformed in a way consistent with the renaming/mapping.

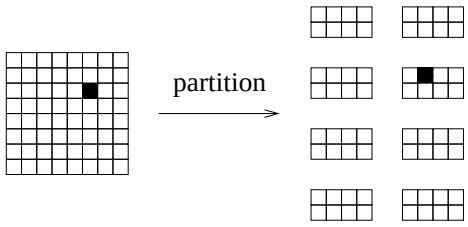


Figure 3.1: Partitioning a 16 by 16 array into 8 array sections.

3.3.3 Data distribution: example

Continuing the example of Section 3.1 and Section 3.2, let program P be the following program:

```
integer a(N), b(N), c(N)
arb
    do i = 1, N/2
        b(i) = a(i)
        c(i) = b(i)
    end do
```

```
        do i = N/2 + 1, N
            b(i) = a(i)
            c(i) = b(i)
        end do
    end arb
```

We can effectively partition arrays `a`, `b`, and `c` into two distinct groups of data elements by mapping each 1-dimensional array of size N onto a 2-dimensional array of size $N/2$ by 2, where each column of the 2-dimensional array represents a local section of the partitioned array. Applying this map to program P produces the following equivalent program P':

```
    integer a(N/2, 2), b(N/2, 2), c(N/2, 2)
    arb
        do i = 1, N/2
            b(i, 1) = a(i, 1)
            c(i, 1) = b(i, 1)
        end do
        do i = 1, N/2
            b(i, 2) = a(i, 2)
            c(i, 2) = b(i, 2)
        end do
    end arb
```

3.3.4 Data duplication: definition and argument for correctness

The transformations involved in data duplication are less obviously semantics-preserving than those involved in data distribution. The goal of such a transformation is to replace a single variable with multiple copies, such that "copy consistency is maintained when it matters." We use the term *(re-)establishing copy consistency* to refer to (re-)establishing the property that all of the copies have the same value (and that their value is the same as that of the original variable at an analogous point in the computation). In the transformed program, all copies have the same initial value as the initial value of the original variable (thereby establishing copy consistency), and any reference to a copy that changes its value is followed by program actions to assign the new value to the other copies as well (thereby re-establishing copy consistency when it is violated). Whenever copy consistency holds, a read reference to the original variable can be transformed into a read reference to any one of the copies without changing the meaning of the program.

3.3.4.1 Phase 1: duplicating the variable

We can accomplish such a transformation using the techniques of data refinement, as described in [59]. We begin with the following data-refinement transformation: Given program P with local variables L, duplicating variable w in L means producing a program P' with variables

$$L' = L \setminus \{w\} \cup \{w^{(1)}, \ldots, w^{(N)}\}$$

(where N is the number of copies desired and $w^{(1)}, \ldots, w^{(N)}$ are the copies of w), such that $P \sqsubseteq P'$. It is simplest to think in terms of renaming w to $w^{(1)}$ and then introducing variables $w^{(2)}, \ldots, w^{(N)}$; it is then clear what it means for P' (with variable $w^{(1)}$) to meet the same specification as P (with variable w).

Using the techniques of data refinement, we can produce such a program P' by defining the abstraction invariant

$$\forall j : 2 \leq j \leq N : w^{(j)} = w^{(1)}$$

and transforming P as follows:

- Assign the same initial value to each copy $w^{(j)}$ in $InitL'$ that was assigned to w in $InitL$, and replace any assignment $w := E$ in P with the multiple assignment

  $$w^{(1)}, \ldots, w^{(N)} := E^{(1)}, \ldots, E^{(N)}$$

  where $E^{(k)} = E[w/w^{(j)}]$ (j is arbitrary and can be different for different values of k). Observe that multiple assignment can be implemented as a sequence of assignments, possibly using temporary variables if w affects E.

- Replace any other reference to w in P with a reference to $w^{(j)}$, where j is arbitrary.

The first replacement rule ensures that the abstraction invariant holds after each command; the second rule makes use of the invariant. In our informal terminology, the abstraction invariant states that copy consistency holds, and the two replacement rules respectively (re-)establish and exploit copy consistency.

Let P' be the result of applying these refinement rules to P. Then $P \sqsubseteq P'$. We do not give a detailed proof, but such a proof could be produced using the rules of data refinement (as given in [59]) and structural induction on P.

3.3.4.2 Phase 2: further refinements

For our purposes, however, P' as just defined may not be quite what we want, since in some situations it would be advantageous to postpone re-establishing copy consistency (e.g., it might make it possible

to apply Theorem 3.1, or if there are several duplicated variables, it might be advantageous to defer re-establishing copy consistency until all have been assigned new values), if we can do so without losing the property that $P \sqsubseteq P'$. We observe, then, that

$$
\begin{array}{l}
\quad (w^{(1)}, \ldots, w^{(N)} := E^{(1)}, \ldots, E^{(N)}) \ ; \ Q \\
\sqsubseteq \\
\quad w^{(k)} := E^{(k)} \ ; \ Q \ ; \ (w^{(1)}, \ldots, w^{(k-1)}, w^{(k+1)}, \ldots, w^{(N)} \ := \ w^{(k)}, \ldots, w^{(k)})
\end{array}
$$

as long as for all $j \neq k$, $w^{(j)}$ is not among the variables read or written by Q. The argument for the correctness of this claim is similar to that used to prove Theorem 2.25 in Section 2.2.5.

3.3.4.3 Application to arb-model programs

We can thus give the following replacement rules for duplicating variable w in an **arb**-model program:

- Replace $w := E$ with

$$\mathbf{arb}(w^{(1)} \ := \ E[w/w^{(1)}] \ , \ \ldots \ , \ w^{(N)} := E[w/w^{(N)}]) \ .$$

- If w is not written by any of $P_1, \ldots, P_N$, replace $\mathbf{arb}(P_1, \ldots, P_N)$ with

$$\mathbf{arb}(P_1[w/w^{(1)}], \ldots, P_N[w/w^{(N)}]) \ .$$

- If w is written by P_k but neither read nor written by any other P_k, replace $\mathbf{arb}(P_1, \ldots, P_N)$ with

$$
\begin{array}{l}
\mathbf{arb}(P_1, \ldots, P_k[w/w^{(k)}], \ldots, P_N) \ ; \\
\mathbf{arb}(w^{(1)} := w^{(k)}, \ \ldots, \ w^{(k-1)} := w^{(k)}, \ w^{(k+1)} := w^{(k)}, \ \ldots, \ w^{(N)} := w^{(k)}) \ .
\end{array}
$$

3.3.5 Data duplication: examples

3.3.5.1 Duplicating constants

This example illustrates duplicating a variable whose intended use is as a constant — that is, its value is to be computed once at the beginning of the program and used but not changed thereafter. Duplicating such a variable is appropriate in transforming a program for eventual execution on a distributed-memory architecture. Let program P be the following program:

```
real PI
real b1, b2, f, arccos
```

```
PI = arccos(-1.0)
arb
    b1 = f(PI, 1)
    b2 = f(PI, 2)
end arb
```

Then P is refined by the following program P':

```
real PI1, PI2
real b1, b2, f, arccos
arb
    PI1 = arccos(-1.0)
    PI2 = arccos(-1.0)
end arb
arb
    b1 = f(PI1, 1)
    b2 = f(PI2, 2)
end arb
```

We can then apply Theorem 3.1 to produce the following program P'', which refines P' and thus P:

```
real PI1, PI2
real b1, b2, f, arccos
arb
    seq
        PI1 = arccos(-1.0) ; b1 = f(PI1, 1)
    end seq
    seq
        PI2 = arccos(-1.0) ; b2 = f(PI2, 2)
    end seq
end arb
```

3.3.5.2 Duplicating loop counters

This example illustrates duplicating a loop counter; again, such a duplication is appropriate in transforming a program for eventual execution on a distributed-memory architecture. Let program P be the following program to compute the sum and product of the integers from 1 to N:

```
integer N, j, sum, prod
```

```
arb
    sum = 0
    prod = 1
end arb
do j = 1, N
    arb
        sum = sum + j
        prod = prod * j
    end arb
end do
```

We first rewrite P to make the operations on the loop counter explicit:

```
integer N, j, sum, prod
arb
    sum = 0
    prod = 1
end arb
j = 1
do while (j <= N)
    arb
        sum = sum + j
        prod = prod * j
    end arb
    j = j + 1
end do
```

We can now apply data duplication to produce the following program P', which refines P:

```
integer N, j1, j2, sum, prod
arb
    sum = 0
    prod = 1
end arb
arb
    j1 = 1
    j2 = 1
end arb
```

```
do while (j1 <= N)
    arb
        sum = sum + j1
        prod = prod * j2
    end arb
    arb
        j1 = j1 + 1
        j2 = j2 + 1
    end arb
end do
```

We can apply Theorem 3.1 to produce a further refinement:

```
integer N, j1, j2, sum, prod
arb
    seq
        sum = 0 ; j1 = 1
    end seq
    seq
        prod = 1 ; j2 = 1
    end seq
end arb
do while (j1 <= N)
    arb
        seq
            sum = sum + j1 ; j1 = j1 + 1
        end seq
        seq
            prod = prod * j2 ; j2 = j2 + 1
        end seq
    end arb
end do
```

We observe that `j1 = j2` is an invariant of the loop, and that it is reasonable to suppose that the above program could be further refined to produce the following:

```
integer N, j1, j2, sum, prod
arb
```

```
        seq
            sum = 0 ; j1 = 1
            do while (j1 <= N)
                sum = sum + j1 ; j1 = j1 + 1
            end do
        end seq
        seq
            prod = 1 ; j2 = 1
            do while (j2 <= N)
                prod = prod * j2 ; j2 = j2 + 1
            end do
        end seq
    end arb
```

Such a transformation is a special case of the general transformation for parallel composition and repetition discussed in Section 4.3.1, so we do not give a proof here, but simply observe that the correctness of the above transformation could be proved by the technique of examining and rearranging possible computations used to prove Theorem 2.15 in Section 2.2.2.

3.3.5.3 Creating shadow copies of variables

Ideally, the partitioning of data in a data-distribution scheme allows computation to also be partitioned such that each element of the computational partition addresses only data from the corresponding element of the data partition. This is not always possible, however, so what is typically done is to partition the computation based on the data partition and an *owner-computes* rule (in which process i performs any computation needed to assign new values to variables in the i-th element of the data partition). In this situation, an element of the computational partition may require read access to variables outside its element of the data partition. A technique frequently employed in programs for distributed-memory architectures is to create *shadow copies* of such variables. If the variables involved are boundary values for local sections of an array that has been partitioned and distributed, it is common to dimension the array's local section to include a *ghost boundary* to be used to hold the shadow copies. Program correctness is maintained by updating the value of the shadow copies whenever the value of the main copy changes. As noted previously (Section 3.3.4), however, the timing of the update is somewhat flexible, provided the copies are updated before being used.

This example illustrates such a situation. The computation is a timestep loop in which each step involves the computation of values for elements of array `new` based on values of elements of array `old`, followed by the copying of values from `new` to `old`. Let program P be the following program:

```
integer N, NSTEPS
real old(0:N+1), new(1:N)
integer k
! initialize old(0), old(N+1) to 1.0, other old(i) to 0.0
call initialize(old)
do k = 1, NSTEPS
    arball (i = 1 : N)
        new(i) = 0.5 * (old(i-1) + old(i+1))
    end arball
    arball (i = 1 : N)
        old(i) = new(i)
    end arball
end do
```

We can transform P for eventual execution on a distributed-memory architecture by partitioning arrays `old` and `new` as follows. (For simplicity, we show a transformation for 2 processes; the more general transformation for P processes is similar.) `new` is partitioned into two local sections of equal size $N/2$; elements of `new` are mapped one-to-one to elements of the local sections. `old` is partitioned into two local sections of equal size $(N/2)+2$, with each local section extended on one side by a ghost boundary of width 1. The situation for array `old` is illustrated by Figure 3.2: Elements other than `old(N/2)` and `old((N/2)+1)` are mapped one-to-one to elements of the local sections; elements `old(N/2)` and `old((N/2)+1)` are duplicated, with one copy (the one shaded in Figure 3.2) the shadow copy. As discussed previously, program correctness is maintained as long as copy consistency (between the shadow copies and the elements of which they are duplicates) is (re-)established before being exploited. The following program P' is the result of applying to P this transformation (data

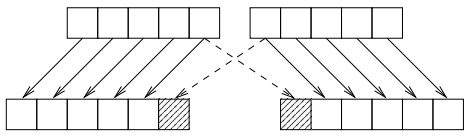


Figure 3.2: Partitioning an array and creating shadow copies.

distribution/duplication), together with a change-of-granularity transformation based on Theorem 3.2:

```
integer N, NSTEPS
real old(0:(N/2)+1, 2), new(1:(N/2), 2)
integer k, i1, i2
```

```
! initialize old(0, 1), old((N/2)+1, 2) to 1.0, other old(i, j) to 0.0
call initialize(old)
do k = 1, NSTEPS
    ! re-establish copy consistency
    arb
        old((N/2)+1, 1) = old(1, 2)
        old(0, 2) = old(N/2, 1)
    end arb
    arb
        do i1 = 1, N/2
            new(i1, 1) = 0.5 * (old(i1-1, 1) + old(i1+1, 1))
        end do
        do i2 = 1, N/2
            new(i2, 2) = 0.5 * (old(i2-1, 2) + old(i2+1, 2))
        end do
    end arb
    arb
        do i1 = 1, N/2
            old(i1, 1) = new(i1, 1)
        end do
        do i2 = 1, N/2
            old(i2, 2) = new(i2, 2)
        end do
    end arb
end do
```

3.3.5.4 Redistributing a variable

In some computations, calculations best performed with one data-distribution scheme are sequentially composed with calculations best performed with a different data-distribution scheme. Examples include the spectral-methods computations described in Section 7.2.2, which are characterized by row operations (performing a calculation on each row of a 2-dimensional array — best performed with data distributed by rows) alternating with column operations (performing a calculation on each column — best performed with data distributed by columns). For such computations, what is typically done is to employ more than one data-distribution scheme and redistribute the data as needed. This strategy can be regarded as an extreme form of data duplication, in which all elements of the array are duplicated, and re-establishing copy consistency involves copying (redistributing) the

entire array. Section 6.1 presents an example of such a computation and how it can be transformed.

3.4 Other transformations

3.4.1 Reductions

If *op* is an associative binary operator over domain D with identity element *ident*, we can define the reduction r of a finite of set of elements $\{d_1, \ldots, d_N\}$ of elements of D with *op* thus:

$$r = d_1 \ op \ \ldots \ op \ d_N$$

We can compute r sequentially with the following program P:

```
r = <ident>
do i = 1, N
    r = r <op> d(i)
end do
```

Such a program cannot be trivially transformed into a program making use of **arb** composition, but we observe that since *op* is associative, it can be refined by the following program P':

```
arb
    seq
        r1 = <ident>
        do i1 = 1, N/2
            r1 = r1 <op> d(i1)
        end do
    end seq
    seq
        r2 = <ident>
        do i2 = N/2 + 1, N
            r2 = r2 <op> d(i2)
        end do
    end seq
end arb
r = r1 <op> r2
```

P' is likely to be more efficient than P when executed on a parallel architecture, assuming that the benefit resulting from dividing the computation between two threads is not overwhelmed by the cost of thread creation.

Examples of operators to which this technique can be applied are integer addition and multiplication (assuming no overflow), and finding the minimum or maximum. Floating-point addition and multiplication are not in general associative and so cannot be treated in this manner unless it is acceptable to ignore discrepancies arising from their lack of associativity; whether this acceptable may depend on both the application and the data being summed or multiplied.

3.4.2 *skip* as an identity element

Given that *skip* is an identity element for sequential composition, it is also an identity element for **arb** composition.

Theorem 3.3.

$$P \sim \mathbf{arb}(skip, P)$$

□

Proof of Theorem 3.3.

Trivial.

□

This theorem can be useful in padding an **arb** composition to take advantage of Theorem 3.1, as in the following example. Let program P be the following:

```
arb
    a1 = 1
    a2 = 2
end arb
b = 10
arb
    c1 = a1
    c2 = a2
end arb
```

We can apply Theorem 3.3 and Theorem 3.1 to get the following refinement of P:

```
arb
    seq
        a1 = 1 ; b = 10 ; c1 = a1
    end seq
    seq
        a2 = 2 ; c2 = a2
    end seq
end arb
```

Chapter 4

The par model and shared-memory programs

As discussed in Chapter 1, once we have developed a program in our **arb** model, we can transform the program into one suitable for execution on a shared-memory architecture via what we call the ***par** model*, which is based on a structured form of parallel composition with barrier synchronization that we call ***par** composition*. In our methodology, we initially write down programs using **arb** composition and sequential constructs; after applying transformations such as those presented in Chapter 3, we transform the results in **par**-model programs, which are then readily converted into programs for shared-memory architectures (by replacing **par** composition with parallel composition and our barrier synchronization construct with that provided by a selected parallel language or library). As noted in Chapter 2, **arb**-model programs can be executed directly on shared-memory architectures, but they may not be very efficient, particularly if the cost of thread creation is high. **par**-model programs are more likely to be efficient for such architectures, and in addition serve as an intermediate stage in the process of transforming **arb**-model programs into programs for distributed-memory architectures. In this chapter we address the following topics:

- Extending our model of parallel composition to include barrier synchronization.
- Transforming **arb**-model programs into programs using parallel composition with barrier synchronization.
- Executing such programs on shared-memory architectures.

4.1 Parallel composition with barrier synchronization

We first expand the definition of parallel composition given in Chapter 2 (Definition 2.12) to include barrier synchronization. Behind any synchronization mechanism is the notion of "suspending" a component of a parallel composition until some condition is met — that is, temporarily interrupting the normal flow of control in the component, and then resuming it when the condition is met. We model suspension as busy waiting, since this approach simplifies our definitions and proofs by making it unnecessary to distinguish between computations that terminate normally and computations that terminate in a deadlock situation — if suspension is modeled as a busy wait, deadlocked computations are infinite.

4.1.1 Specification of barrier synchronization

We first give a specification for barrier synchronization; that is, we define the expected behavior of a *barrier command* in the context of the parallel composition of programs $P_1, \ldots, P_N$. If iB_j denotes the number of times P_j has initiated the barrier command, and cB_j denotes the number of times P_j has completed the barrier command, then we require the following:

- For all j, $iB_j = cB_j$ or $iB_j = cB_j + 1$. If $iB_j = cB_j + 1$, we say that P_j is suspended at the barrier. If $iB_j = cB_j$, we say that P_j is not suspended at the barrier.
- If P_j and P_k are both suspended at the barrier, or neither P_j nor P_k is suspended at the barrier, then $iB_j = iB_k$.
- If P_j is suspended at the barrier and P_k is not suspended at the barrier, $iB_j = iB_k + 1$.
- For any n, if every P_j initiates the barrier command n times, then eventually every P_j completes the barrier command n times:

$$(\forall j :: (iB_j = cB_j + 1) \wedge (iB_j = n)) \leadsto (\forall j :: (cB_j = n)) \ .$$

We observe that this specification simply captures formally the usual meaning of barrier synchronization and is consistent with other formalizations, for example those of [2] and [70]. Most details of the specification were obtained from [72]; the overall method (in which initiations and completions of a command are considered separately) owes much to [55].

4.1.2 Definitions

We define barrier synchronization by extending the definition of parallel composition given in Definition 2.12 and defining a new command, **barrier**. This combined definition implements a common

approach to barrier synchronization based on keeping a count of processes waiting at the barrier, as in [2] and [70]. In the context of our model, we implement this approach using two protocol variables local to the parallel composition, a count Q of suspended components and a flag *Arriving* that indicates whether components are arriving at the barrier or leaving. As components arrive at the barrier, we suspend them and increment Q. When Q equals the number of components, we set *Arriving* to *false* and allow components to leave the barrier. Components leave the barrier by unsuspending and decrementing Q. When Q equals 0, we reset *Arriving* to *true*, ready for the next use of the barrier.

Definition 4.1 (barrier).

We define program **barrier** $= (V, L, InitL, A, PV, PA)$ as follows:

- $V = L \cup \{Q, Arriving\}$.
- $L = \{En, Susp\}$, where $En, Susp$ are Boolean variables.
- $InitL = (true, false)$.
- $A = \{a_{arrive}, a_{release}, a_{leave}, a_{reset}, a_{wait}\}$, where
  - a_{arrive} corresponds to a process's initiating the barrier command when fewer than $N-1$ other processes are suspended. The process should then suspend, so the action is defined by the set of state transitions $s \to s'$ such that:
    - In s, *En* is *true*, *Arriving* is *true*, and $Q < (N-1)$.
    - s' is s with *En* set to *false*, *Susp* set to *true*, and Q incremented by 1.
  - $a_{release}$ corresponds to a process's initiating the barrier command when $N-1$ other processes are suspended. The process should then complete the command and enable the other processes to complete their barrier commands as well. The action is thus defined by the set of state transitions $s \to s'$ such that:
    - In s, *En* is *true*, *Arriving* is *true*, and $Q = (N-1)$.
    - s' is s with *En* set to *false* and *Arriving* set to *false*. *Susp*, which was initially *false*, is unchanged.
  - a_{leave} corresponds to a process's completing the barrier command when at least one other process has not completed its barrier command. The action is defined by the set of state transitions $s \to s'$ such that:
    - In s, *Susp* is *true*, *Arriving* is *false*, and $Q > 1$.
    - s' is s with *Susp* set to *false* and Q decremented by 1.

  - a_{reset} corresponds to a process's completing the barrier command when all other processes have already done so. The action is defined by the set of state transitions $s \rightarrow s'$ such that:
    * In s, *Susp* is *true*, *Arriving* is *false*, and $Q = 1$.
    * s' is s with *Susp* set to *false*, *Arriving* set to *true*, and Q set to 0.
  - a_{wait} corresponds to a process's busy-waiting at the barrier. The action is defined by the set of state transitions $s \rightarrow s'$ such that:
    * In s, *Susp* is *true*.
    * $s' = s$.

- $PV = \{Q, Arriving\}$.
- $PA = A$.

□

Definition 4.2 (Parallel composition with barrier synchronization).

We define parallel composition as in Chapter 2 (Definition 2.12), except that we add local protocol variables *Arriving* (of type Boolean) and Q (of type integer) with initial values *true* and 0 respectively.

□

Remarks about Definition 4.2.

- This definition meets the specification given in Section 4.1.1; a proof can be constructed by formalizing the introductory discussion of Section 4.1.2. Observe that the last point of the specification — the required progress property — is in part a consequence of our fairness requirement for computations.

□

4.2 The par model

We now define a structured form of parallel composition with barrier synchronization. Previously we defined a notion of **arb**-compatibility and then defined **arb** composition as the parallel composition

of **arb**-compatible components. Analogously, in this chapter we define a notion of **par**-compatibility and then define **par** composition as the parallel composition of **par**-compatible components. The idea behind **par**-compatibility is that the components match up with regard to their use of the barrier command — that is, they all execute the barrier command the same number of times and hence do not deadlock.

4.2.1 Preliminary definitions

Definition 4.3 (Free barrier).

Program P is said to contain a *free barrier* exactly when it contains an instance of **barrier** not enclosed in a parallel composition.

□

Examples of Definition 4.3.

Q; **barrier**; R contains a free barrier. $(Q_1; \textbf{barrier}; R_1)||(Q_1; \textbf{barrier}; R_1)$ does not.

□

Definition 4.4 (arb-compatible, revisited).

Programs $P_1, \ldots, P_N$ are **arb**-compatible exactly when (1) they meet the conditions for **arb**-compatibility given earlier (Definition 2.14), and (2) for each j, P_j contains no free barriers.

□

4.2.2 par-compatibility

We can now define **par**-compatibility. Observe that this definition is given in terms of restricted forms of the alternative (*IF*) and repetition (*DO*) constructs of Dijkstra's guarded-command language [35, 37], but it applies to any programming notation with equivalent constructs.

Definition 4.5 (par-compatible).

We say programs $P_1, \ldots, P_N$ are **par**-compatible exactly when one of the following is true:

- $P_1, \ldots, P_N$ are **arb**-compatible.

- For each j,

$$P_j = Q_j; \mathbf{barrier}; R_j$$

where $Q_1, \ldots, Q_N$ are **arb**-compatible and $R_1, \ldots, R_N$ are **par**-compatible.

- For each j,

$$P_j = \mathbf{if}\ b_j\ \rightarrow\ Q_j\ []\ \neg b_j\ \rightarrow\ \mathit{skip}\ \mathbf{fi}$$

where $Q_1, \ldots, Q_N$ are **par**-compatible, and for $k \neq j$ no variable that affects b_j is written by Q_k.

- For each j,

$$P_j = \mathbf{if}\ b_j\ \rightarrow\ (Q_j; \mathbf{barrier}; R_j)\ []\ \neg b_j\ \rightarrow\ \mathit{skip}\ \mathbf{fi}$$

where $Q_1, \ldots, Q_N$ are **arb**-compatible, $R_1, \ldots, R_N$ are **par**-compatible, and for $k \neq j$ no variable that affects b_j is written by Q_k.

- For each j,

$$P_j = \mathbf{do}\ b_j\ \rightarrow\ (Q_j; \mathbf{barrier}; R_j; \mathbf{barrier})\ \mathbf{od}$$

where $Q_1, \ldots, Q_N$ are **arb**-compatible, $R_1, \ldots, R_N$ are **par**-compatible, and for $k \neq j$ no variable that affects b_j is written by Q_k.

□

4.2.3 par composition

As with **arb**, we write $\mathbf{par}(P_1, \ldots, P_N)$ to denote the parallel composition (with barrier synchronization) of **par**-compatible elements $P_1, \ldots, P_N$.

4.2.3.1 Fortran 90 notation

Again as for **arb**, we define a slightly different notation for use with Fortran 90. As for **arb**, this notation allows us to develop programs using the **arb** and **par** models that can be easily transformed into programs in practical languages based on Fortran 90, as described in Section 4.4. For **par**-compatible programs $P_1, \ldots, P_N$, we write their **par** composition thus:

```
par
    P_1
    P_2
    ...
    P_N
end par
```

4.2.3.2 parall

We also define a syntax **parall** analogous to **arball**.

Definition 4.6 (parall).

If we have N index variables $i_1, \ldots, i_N$, with corresponding index ranges $i_j\_start \leq i_j \leq i_j\_end$, and program block P such that P does not modify the value of any of the index variables — that is, $\mathbf{mod}.P \cap \{i_1, \ldots, i_N\} = \{\}$ — then we can define an **parall** composition as follows:

For each tuple $(x_1, \ldots, x_N)$ in the cross product of the index ranges, we define a corresponding program block $P(x_1, \ldots, x_N)$ by replacing index variables $i_1, \ldots, i_N$ with corresponding values $x_1, \ldots, x_N$. If the resulting program blocks are **par**-compatible, then we write their **par** composition as follows:

```
parall (i_1 = i_1_start : i_1_end , ..., i_N = i_N_start : i_N_end)
    P(x_1, ..., x_N)
end parall
```

□

Remarks about Definition 4.6.

- As for **arball** (Definition 2.27), the body of the **parall** composition can be a sequential composition. We do not require that the sequential composition be explicit, as illustrated in the next-to-last example.

□

4.2.4 Examples of par composition

Composition of sequential blocks

The following example composes two sequences, the first assigning to `a` and `b` and the second assigning to `c` and `d`. (Here, the barrier is not needed, and is included purely as an illustration of a syntactically valid use.)

```
par
    seq
        a = 1 ; barrier ; b = a
    end seq
    seq
        c = 2 ; barrier ; d = c
    end seq
end par
```

Composition of sequential blocks (parall)

The following example composes ten sequences, each assigning to one element of `a` and one element of `b`. Here, the barrier is needed, since otherwise the sequences being composed would not be **par**-compatible.

```
parall (i = 1:10)
    seq
        a(i) = i
        barrier
        b(i) = a(11-i)
    end seq
end parall
```

As noted in the remarks following Definition 4.6, if the body of the **parall** composition is a sequential composition, we do not require that the sequential composition be explicit; that is, this example could also be written:

```
parall (i = 1:10)
    a(i) = i
    barrier
    b(i) = a(11-i)
end parall
```

without changing its meaning.

Invalid composition

The following example is not a valid **par** composition; the two sequences are not **par**-compatible.

```
par
    seq
        a = 1 ; barrier ; b = a
    end seq
    seq
        c = 2
    end seq
end par
```

4.3 Transforming arb-model programs into par-model programs

We now give theorems allowing us to transform programs in the **arb** model into programs in the **par** model.

4.3.1 Theorems

Theorem 4.7 (Replacement of arb with par).

If $P_1, \ldots, P_N$ are **arb**-compatible,

$$\mathbf{arb}(P_1, \ldots, P_N) \sqsubseteq \mathbf{par}(P_1, \ldots, P_N)$$

□

Proof of Theorem 4.7.

Trivial.

□

Theorem 4.8 (Interchange of par and sequential composition).

If $Q_1, \ldots, Q_N$ are **arb**-compatible and $R_1, \ldots, R_N$ are **par**-compatible, then

$$\begin{array}{l} \quad \mathbf{arb}(Q_1, \ldots, Q_N); \mathbf{par}(R_1, \ldots, R_N) \\ \sqsubseteq \\ \quad \mathbf{par}(\\ \qquad (Q_1; \mathbf{barrier}; R_1), \\ \qquad \ldots, \\ \qquad (Q_N; \mathbf{barrier}; R_N) \\ \quad) \end{array}$$

□

Proof of Theorem 4.8.

First observe that both sides of the refinement have the same set of non-local variables V_{nl}. We need to show that given any maximal computation C of the right-hand side of the refinement we can produce a maximal computation C' of the left-hand side such that C' is equivalent to C with respect to V_{nl}. This is straightforward: In any maximal computation of the right-hand side, from the definitions of sequential composition and **barrier** we know that we can partition the computation into (1) a segment consisting of maximal computations of the Q_j's and initiations of the **barrier** command, one for each j, and (2) a segment consisting of completions of the **barrier** command, one for each j, and maximal computations of the R_j's. Segment (1) can readily be mapped to an equivalent maximal computation of $\mathbf{arb}(Q_1, \ldots, Q_N)$ by removing the barrier-initiation actions. Segment (2) can readily be mapped to an equivalent maximal computation of $\mathbf{par}(R_1, \ldots, R_N)$ by removing the first barrier-completion action for each j. We observe that this approach works even for nonterminating computations: If the right-hand side does not terminate, then either at least one Q_j does not terminate, or $\mathbf{par}(R_1, \ldots, R_N)$ does not terminate, and in either case the analogous computation of the left-hand side also does not terminate. The right-hand side cannot fail to terminate because of deadlock at the first barrier because if all the Q_j's terminate, the immediately-following executions of **barrier** terminate as well (from the specification of barrier synchronization).

□

Theorem 4.9 (Interchange of par and *IF*, part 1).

If $Q_1, \ldots, Q_N$ are **par**-compatible, and for all j no variable that affects b is written by Q_j, then

if $b \rightarrow$ **par**$(Q_1, \ldots, Q_N)$ [] $\neg b \rightarrow$ *skip* **fi**

$\sqsubseteq$

par(
  if $b \rightarrow Q_1$ [] $\neg b \rightarrow$ *skip* **fi**,
  $\ldots$,
  if $b \rightarrow Q_N$ [] $\neg b \rightarrow$ *skip* **fi**
)

□

Proof of Theorem 4.9.

Again observe that both sides of the refinement have the same set of non-local variables V_{nl}. As before, a proof can be constructed by considering all maximal computations of the right-hand side and showing that for each such computation C we can produce a maximal computation C' of the left-hand side such that C' is equivalent to C with respect to V_{nl}. Here, such a proof uses the fact that the value of b is not changed by Q_j for any j. Since no barriers are introduced in this transformation, we do not introduce additional possibilities for deadlock.

□

Lemma 4.10 (Interchange of par and *IF*, part 1, with duplicated variables).

If $Q_1, \ldots, Q_N$ and b are as for Theorem 4.9, and $b_1, \ldots, b_N$ are Boolean expressions such that for $j \neq k$ no variable that affects b_j is written by Q_k, then the following holds whenever both sides are started in a state in which $b_j = b$ for all j:

if b $\rightarrow$ **par**$(Q_1, \ldots, Q_N)$ [] $\neg b$ $\rightarrow$ *skip* **fi**

$\sqsubseteq$

par(
  if b_1 $\rightarrow$ Q_1 [] $\neg b_1$ $\rightarrow$ *skip* **fi**,
  $\ldots$,
  if b_N $\rightarrow$ Q_N [] $\neg b_N$ $\rightarrow$ *skip* **fi**
)

□

Proof of Lemma 4.10.

This lemma follows from Theorem 4.9 and exploitation of copy consistency as discussed in Section 3.3.4.

□

Theorem 4.11 (Interchange of par and *IF*, part 2).

If $Q_1, \ldots, Q_N$ are **arb**-compatible, $R_1, \ldots, R_N$ are **par**-compatible, and for all j no variable that affects b is written by Q_j, then

if b $\rightarrow$ (**arb**$(Q_1, \ldots, Q_N)$; **par**$(R_1, \ldots, R_N)$) [] $\neg b$ $\rightarrow$ *skip* **fi**

$\sqsubseteq$

par(
  if b $\rightarrow$ (Q_1; **barrier**; R_1) [] $\neg b$ $\rightarrow$ *skip* **fi**,
  $\ldots$,
  if b $\rightarrow$ (Q_N; **barrier**; R_N) [] $\neg b$ $\rightarrow$ *skip* **fi**
)

□

Proof of Theorem 4.11.

Again observe that both sides of the refinement have the same set of non-local variables V_{nl}. As before, a proof can be constructed by considering all maximal computations of the right-hand side and showing that for each such computation C we can produce a maximal computation C' of the left-hand side such that C' is equivalent to C with respect to V_{nl}. The barrier introduced in the transformation cannot deadlock for reasons similar to those for the transformation of Theorem 4.8.

□

Lemma 4.12 (Interchange of par and *IF*, part 2, with duplicated variables).

If $Q_1, \ldots, Q_N$, $R_1, \ldots, R_N$, and b are as for Theorem 4.11, and $b_1, \ldots, b_N$ are Boolean expressions such that for $j \neq k$ no variable that affects b_j is written by Q_k, then the following holds whenever both sides are started in a state in which $b_j = b$ for all j:

if b $\rightarrow$ (**arb**$(Q_1, \ldots, Q_N)$; **par**$(R_1, \ldots, R_N)$) [] $\neg b$ $\rightarrow$ *skip* **fi**

$\sqsubseteq$

par(
  if b_1 $\rightarrow$ (Q_1; **barrier**; R_1) [] $\neg b_1$ $\rightarrow$ *skip* **fi**,
  $\ldots$,
  if b_N $\rightarrow$ (Q_N; **barrier**; R_N) [] $\neg b_N$ $\rightarrow$ *skip* **fi**
)

□

Proof of Lemma 4.12.

Analogous to Lemma 4.10.

□

Theorem 4.13 (Interchange of par and *DO*).

If $Q_1, \ldots, Q_N$ are **arb**-compatible, $R_1, \ldots, R_N$ are **par**-compatible, and for all j no variable that affects b is written by Q_j, then

$$
\begin{array}{l}
\quad \textbf{do } b \ \rightarrow \ (\textbf{arb}(Q_1, \ldots, Q_N); \textbf{par}(R_1, \ldots, R_N)) \textbf{ od} \\
\sqsubseteq \\
\quad \textbf{par}(\\
\qquad \textbf{do } b \ \rightarrow \ (Q_1; \textbf{barrier}; R_1; \textbf{barrier}) \textbf{ od}, \\
\qquad \ldots, \\
\qquad \textbf{do } b \ \rightarrow \ (Q_N; \textbf{barrier}; R_N; \textbf{barrier}) \textbf{ od} \\
\quad)
\end{array}
$$

□

Proof of Theorem 4.13.

First observe that both sides of the refinement have the same set of non-local variables V_{nl}. As before, a proof can be constructed by considering all maximal computations of the right-hand side and showing that for each such computation C we can produce a maximal computation C' of the left-hand side such that C' is equivalent to C with respect to V_{nl}. The proof makes use of the restrictions on when variables that affect b can be written. For terminating computations, the proof can be constructed using the standard unrolling of the repetition command (as in [42] or [37]) together with Theorem 4.8 and Theorem 4.11. For nonterminating computations, the proof must consider two classes of computations: those that fail to terminate because an iteration of one of the loops fails to terminate, and those that fail to terminate because one of the loops iterates forever. In both cases, however, the computation can be mapped onto an infinite (and therefore, in our model, equivalent) computation of the left-hand side.

□

Lemma 4.14 (Interchange of par and DO, with duplicated variables).

If $Q_1, \ldots, Q_N$ are **arb**-compatible, $R_1, \ldots, R_N$ are **par**-compatible, and for all $k \neq j$ no variable that affects b_j is written by Q_k, and $(\forall j :: (b_j = b))$ is an invariant of the loop

$$\textbf{do } b \ \rightarrow \ (\textbf{arb}(Q_1, \ldots, Q_N); \textbf{par}(R_1, \ldots, R_N)) \textbf{ od}$$

then the following holds whenever both sides are started in a state in which $b_j = b$ for all j:

do $b \ \rightarrow \ (\mathbf{arb}(Q_1, \ldots, Q_N); \mathbf{par}(R_1, \ldots, R_N))$ **od**

$\sqsubseteq$

par(

do $b_1 \ \rightarrow \ (Q_1; \mathbf{barrier}; R_1, \mathbf{barrier})$ **od**,

$\ldots$,

do $b_N \ \rightarrow \ (Q_N; \mathbf{barrier}; R_N, \mathbf{barrier})$ **od**

)

□

Proof of Lemma 4.14.

Analogous to Lemma 4.10.

□

4.3.2 Examples

Replacing arb with par (Theorem 4.7)

Let P be the following program:

```
arball (i = 1 : 10)
    a(i) = i
end arball
```

Then P is refined by the following:

```
parall (i = 1 : 10)
    a(i) = i
end parall
```

Interchanging par and sequential composition (Theorem 4.8)

Let P be the following program:

```
arb
    new(1) = old(1) + 0.5*old(2)
    new(2) = old(2) + 0.5*old(1)
```

```
end arb
arb
    old(1) = new(1)
    old(2) = new(2)
end arb
```

Then P is refined by the following:

```
par
    seq
        new(1) = old(1) + 0.5*old(2)
        barrier
        old(1) = new(1)
    end seq
    seq
        new(2) = old(2) + 0.5*old(1)
        barrier
        old(2) = new(2)
    end seq
end par
```

Interchanging par and *IF* (Theorem 4.9)

Let P be the following program:

```
if (x > 0) then
    par
        a = 1
        b = 2
    end par
end if
```

Then P is refined by the following:

```
par
    if (x > 0) then
        a = 1
    end if
    if (x > 0) then
```

```
            b = 2
        end if
    end par
```

Interchanging par and *IF* (Theorem 4.11)

Let P be the following program:

```
    if (x > 0) then
        arb
            a = 1
            b = 2
        end arb
        par
            x = x + 1
            skip
        end par
    end if
```

Then P is refined by the following:

```
    par
        if (x > 0) then
            a = 1 ; barrier ; x = x + 1
        end if
        if (x > 0) then
            b = 2 ; barrier ; skip
        end if
    end par
```

Interchanging par and *DO* (Theorem 4.13)

Let P be the following program:

```
    do while (x < 100)
        arb
            a = a * 2
            b = b + 1
        end arb
        par
```

```
            x = max(a, b)
            skip
        end par
    end do
```

Then P is refined by the following:

```
    par
        do while (x < 100)
            a = a * 2 ; barrier ; x = max(a, b) ; barrier
        end do
        do while (x < 100)
            b = b + 1 ; barrier ; skip ; barrier
        end do
    end par
```

Interchanging par and *DO* (Theorem 4.14)

Let P be the following program:

```
    x = max(a, b)
    do while (x < 100)
        arb
            a = a * 2
            b = b + 1
        end arb
        par
            x = max(a, b)
            skip
        end par
    end do
```

Then P is refined (using the data-duplication techniques of Section 3.3) by the following:

```
    arb
        x1 = max(a, b)
        x2 = max(a, b)
    end arb
    do while (x1 < 100)
```

```
        arb
            a = a * 2
            b = b + 1
        end arb
        par
            x1 = max(a, b)
            x2 = max(a, b)
        end par
    end do
```

which in turn is refined (using Theorem 4.14) by the following:

```
    arb
        x1 = max(a, b)
        x2 = max(a, b)
    end arb
    par
        do while (x1 < 100)
            a = a * 2 ; barrier ; x1 = max(a, b) ; barrier
        end do
        do while (x2 < 100)
            b = b + 1 ; barrier ; x2 = max(a, b) ; barrier
        end do
    end par
```

which again in turn is refined by the following:

```
    par
        seq
            x1 = max(a, b)
            barrier
            do while (x1 < 100)
                a = a * 2 ; barrier ; x1 = max(a, b) ; barrier
            end do
        end seq
        seq
            x2 = max(a, b)
            barrier
```

```
        do while (x2 < 100)
            b = b + 1 ; barrier ; x2 = max(a, b) ; barrier
        end do
    end seq
end par
```

4.4 Executing par-model programs

It is clear that **par** composition as described in this chapter is implemented by general parallel composition (as described in Section 2.6.2) plus a barrier synchronization that meets the specification of Section 4.1.1. Thus, we can transform a program in the **par** model into an equivalent program in any language that implements parallel composition and barrier synchronization in a way consistent with our definitions (which in turn are consistent with the usual meaning of parallel composition with barrier synchronization).

4.4.1 Parallel execution using X3H5 Fortran

For example, a **par**-model program can be transformed into an equivalent program in the notation of the Fortran X3H5 proposal [3] by replacing `par` and `end par` with `PARALLEL SECTIONS`, `SECTION`, and `END PARALLEL SECTIONS`, replacing `parall` and `end parall` with `PARALLEL DO` and `END PARALLEL DO` (nested if necessary), and replacing **barrier** with `BARRIER`.

4.4.2 Example

For example, the last example of Section 4.3.2 is equivalent to the following program segment using the X3H5 extensions to Fortran:

```
PARALLEL SECTIONS
SECTION
    x1 = max(a, b)
    BARRIER
    do while (x1 < 100)
        a = a * 2
        BARRIER
        x1 = max(a, b)
        BARRIER
    end do
SECTION
```

```
    x2 = max(a, b)
    BARRIER
    do while (x2 < 100)
        b = b + 1
        BARRIER
        x2 = max(a, b)
        BARRIER
    end do
END PARALLEL SECTIONS
```

Chapter 5

The subset par model and distributed-memory programs

As discussed in Chapter 1, once we have developed a program in our **arb** model, we can transform the program into one suitable for execution on a distributed-memory–message-passing architecture via what we call the *subset **par** model*, which is a restricted form of the **par** model discussed in Chapter 4. In our methodology, we apply a succession of transformations to an **arb**-model program to produce a program in the subset **par** model and then transform the result into a program for a distributed-memory–message-passing architecture. In this chapter we address the following topics:

- Extending our model of parallel composition to include message-passing operations.
- Restricting the **par** model to correspond more directly to distributed-memory architectures.
- Transforming programs in the resulting subset **par** model into programs using parallel composition with message-passing.
- Executing such programs on distributed-memory–message-passing architectures.

5.1 Parallel composition with message-passing

We first expand the definition of parallel composition given in Chapter 2 to include message-passing.

5.1.1 Specification

We define message-passing for $P_1, \ldots, P_N$ composed in parallel in a way compatible with single-sender–single-receiver channels with infinite slack (i.e., infinite capacity). Every message operation (send or receive) specifies a sender and a receiver, and while a receive operation suspends if there is

no message to receive, a send operation never suspends. Messages are received in the order in which they are sent and are not received before they are sent. That is, if we let $nS_{j,k}$ denote the number of send operations from P_j to P_k performed, $iR_{j,k}$ denote the number of receive operations from P_j to P_k initiated, and $cR_{j,k}$ denote the number of such receive operations completed, then we can write the desired specification as follows:

- $iR_{j,k} = cR_{j,k}$ or $iR_{j,k} = cR_{j,k} + 1$ for all j, k.
- Messages are not received before they are sent: $nS_{j,k} \geq cR_{j,k}$ for all j, k.
- Messages are received in the order in which they are sent: The n-th message received by P_j from P_k is identical with the n-th message sent from P_k to P_j.
- If n messages are sent from P_k to P_j, and P_j initiates n receive operations for messages from P_k, then all will complete:

$$(nS_{j,k} \geq n) \wedge (iR_{j,k} = n)) \rightsquigarrow (cR_{j,k} = n) \quad .$$

We observe that this specification, like the one for barrier synchronization in Chapter 4, simply captures formally the usual meaning of this type of message passing, and is consistent with other formalizations, for example those of [2] and [69]. The terminology ("slack") and overall method (in which initiations and completions of a command are considered separately) are based on [55].

5.1.2 Definitions

Like many other implementations of message-passing, for example those of [2] and [69], our definition represents channels as queues:

We define for each ordered pair (P_j, P_k) a queue $C_{j,k}$ whose elements represent messages in transit from P_j to P_k. Message sends are then represented as enqueue operations and message receives as (possibly suspending) dequeue operations. Elements of $C_{j,k}$ take the form of pairs (*Type*, *Value*). Just as we did in Chapter 4, we model suspension as busy waiting.

Definition 5.1 (send).

We define program **send** $= (V, L, InitL, A, PV, PA)$ as follows:

- $V = L \cup \{OutP_1, \ldots, OutP_N, Rcvr, Type, Value\}$, where each $OutP_j$ ("outport j") is a variable of type queue, *Rcvr* is an integer variable, *Type* is a type, and *Value* is a variable of type *Type*. Variables $OutP_1, \ldots, OutP_N$ are to be shared with the enclosing parallel composition, as described later, while variables *Rcvr*, *Type*, *Value* are to be shared with the enclosing sequential

composition. (I.e., it is assumed that **send** is composed in sequence with assignment statements that assign appropriate values to *Rcvr*, *Type*, and *Value*.)

- $L = \{En\}$, where En is a Boolean variable.
- $InitL = (true)$.
- $A = \{a_{snd}\}$, where
  - a_{snd} corresponds to a process's sending a message (*Type*, *Value*) to process P_{Rcvr}. The action is defined by the set of state transitions $s \rightarrow s'$ such that:
    - In s, En is *true*.
    - s' is s with En set to *false* and (*Type*, *Value*) enqueued (appended) to $OutP_{Rcvr}$.
- $PV = \{OutP_1, \ldots, OutP_N\}$.
- $PA = A$.

□

Definition 5.2 (recv).

We define program **recv** $= (V, L, InitL, A, PV, PA)$ as follows:

- $V = L \cup \{InP_1, \ldots, InP_N, Sndr, Type, Value\}$, where each InP_j ("inport j") is a variable of type queue, *Sndr* is an integer variable, *Type* is a type, and *Value* is a variable of type *Type*. Variables $InP_1, \ldots, InP_N$ are to be shared with the enclosing parallel composition, as described later, while variables *Sndr*, *Type*, *Value* are to be shared with the enclosing sequential composition, similarly to the analogous variables of **send**.
- $L = \{En\}$, where En is a Boolean variable.
- $InitL = (true)$.
- $A = \{a_{rcv}, a_{wait}\}$, where
  - a_{rcv} corresponds to a process's receiving a message (*Type*, *Value*) from process P_{Sndr}. The action is defined by the set of state transitions $s \rightarrow s'$ such that:
    - In s, En is *true* and InP_{Sndr} is not empty.
    - s' is s with En set to *false* and (*Type*, *Value*) and InP_{Sndr} set to the values resulting from dequeueing an element from InP_{Sndr}.

  - a_{wait} corresponds to a process's waiting for a message from process P_{Sndr}. The action is defined by the set of state transitions $s \rightarrow s'$ such that:
    * In s, En is *true* and InP_{Sndr} is empty.
    * $s' = s$.

- $PV = \{InP_1, \ldots, InP_N\}$.
- $PA = A$.

□

Definition 5.3 (Parallel composition with message-passing).

We define parallel composition as in Chapter 2 (Definition 2.12), except that we add local protocol variables $C_{j,k}$ (of type queue), one for each ordered pair (P_j, P_k), with initial values of "empty", and we perform the following additional modifications on the component programs P_j:

- We replace variables $OutP_1, \ldots, OutP_N$ in V_j with $C_{j,1}, \ldots, C_{j,N}$, and we make the same replacement in actions a derived from a_{snd}.
- We replace variables $InP_1, \ldots, InP_N$ in V_j with $C_{1,j}, \ldots, C_{N,j}$, and we make the same replacement in actions a derived from a_{rcv} and a_{wait}.

□

Remarks about Definition 5.3.

- This definition clearly meets the specification given in Section 5.1.1.

□

5.2 The subset par model

We define the subset **par** model such that a computation of a program in this model may be thought of as consisting of an alternating sequence of (1) blocks of computation in which each component operates independently on its local data, and (2) blocks of computation in which values are copied between components, separated by barrier synchronization, as illustrated by Figure 5.1. Shaded vertical bars represent computations of processes, arrows represent copying of data between processes, and dashed horizontal lines represent barrier synchronization. We refer to a block of the first variety as a *local-computation section* and to a block of the second variety (together with the preceding and succeeding barrier synchronizations) as a *data-exchange operation*.

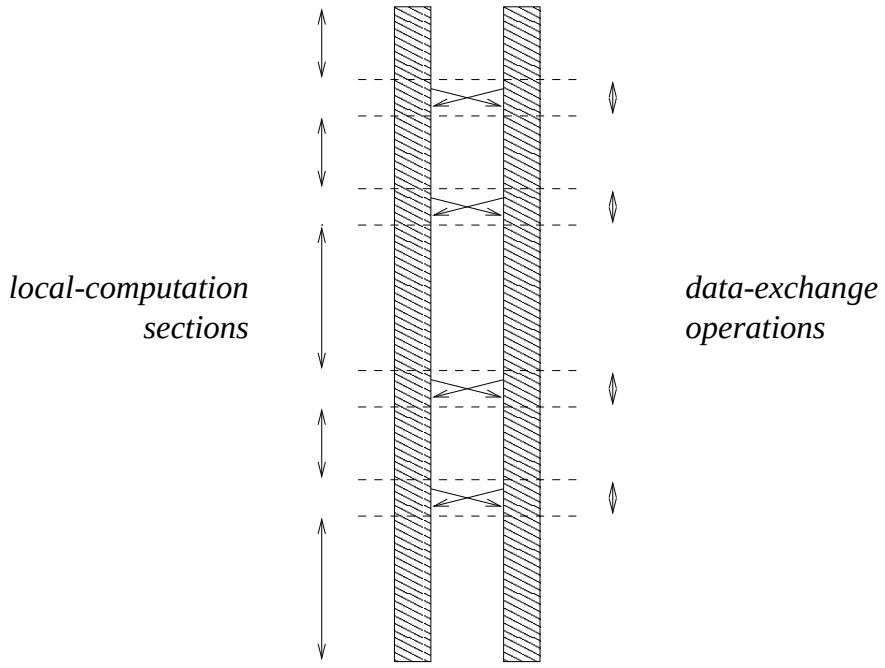


Figure 5.1: A computation of a subset-**par**-model program.

5.2.1 Subset par-compatibility

A program in the subset **par** model is a composition $\mathbf{par}(P_1, \ldots, P_N)$, where $P_1, \ldots, P_N$ are subset-**par**-compatible as defined by the following:

Definition 5.4 (Subset par-compatibility).

$P_1, \ldots, P_N$ are subset-**par**-compatible exactly when (1) $P_1, \ldots, P_N$ are **par**-compatible, (2) the variables V of the composition (excluding the protocol variables representing message channels) are partitioned into disjoint subsets $W_1, \ldots, W_N$, and (3) exactly one of the following holds:

- $P_1, \ldots, P_N$ are **arb**-compatible and each P_j reads and writes only variables in W_j.
- For each j,

  $$P_j = Q_j; \mathbf{barrier}; Q'_j; \mathbf{barrier}; R_j$$

  where
  - $Q_1, \ldots, Q_N$ are **arb**-compatible.
  - Each Q_j reads and writes only variables in W_j.
  - Each Q'_j is an **arb**-compatible set of assignment statements $x_k := x_j$ such that x_j is an element of W_j and x_k is an element of W_k for some k (possibly $k = j$).

  - $R_1, \ldots, R_N$ are subset-**par**-compatible.

- For each j, $b_j \in W_j$ and

$$P_j = \textbf{if } b_j \ \rightarrow \ Q_j \ [] \ \neg b_j \ \rightarrow \ \textit{skip} \ \textbf{fi}$$

  where $Q_1, \ldots, Q_N$ are subset-**par**-compatible.

- For each j, $b_j \in W_j$ and

$$P_j = \textbf{do } b_j \ \rightarrow \ Q_j \ \textbf{od}$$

  where $Q_1, \ldots, Q_N$ are subset-**par**-compatible.

□

5.2.2 Example of subset par composition

Recursive doubling

The following example computes the sum of four elements using recursive doubling:

```
integer a(4), part(2), part_copy(2), m(2)
arb
    part(1) = max(a(1), a(2))
    part(2) = max(a(3), a(4))
end arb
arb
    part_copy(1) = part(2)
    part_copy(2) = part(1)
end arb
arb
    m(1) = max(part(1), part_copy(1))
    m(2) = max(part_copy(2), part(2))
end arb
```

5.3 Transforming subset-par-model programs into programs with message-passing

5.3.1 Transformations

We can transform a program in the subset **par** model into a program for a distributed-memory–message-passing architecture by mapping each component P_j onto a process j and making the

following additional changes:

- Map each element W_j of the partition of V to the address space for process j.
- Convert each data-exchange operation (consisting of a set of (**barrier**; Q'_j; **barrier**) sequences, one for each component P_j) into a collection of message-passing operations, in which each assignment $x_j := x_k$ is transformed into a pair of message-passing commands: a **send** command in k specifying $Rcvr = j$, and a **recv** command in j specifying $Sndr = k$.
- Optionally, for any pair (P_j, P_k) of processes, concatenate all the messages sent from P_j to P_k as part of a data-exchange operation into a single message, replacing the collection of (send, receive) pairs from P_j to P_k with a single (send, receive) pair.

Such a program refines the original program: Each send–receive pair of operations produces the same result as the assignment statement from which it was derived (as discussed in [45] and [56]), and the **arb**-compatibility of the assignments ensures that these pairs can be executed in any order without changing the result. Replacing barrier synchronization with the weaker pairwise synchronization implied by these pairs of message-passing operations also preserves program correctness; we can construct a proof of this claim by using the techniques of Chapter 2 and our definitions of barrier synchronization and message-passing, essentially revising the proof of Theorem 8.1 in Chapter 8 (which predates the development of our operational model) to take advantage of the framework provided by our model for relating the operation of different synchronization mechanisms.

5.3.2 Example

If P is the recursive-doubling example program of Section 5.2.2, P is refined by the following subset-**par**-model program P' with variables partitioned into

- $W_1 = \{\texttt{a(1:2)}, \texttt{part(1)}, \texttt{part\_copy(1)}, \texttt{m(1)}\}$ and
- $W_2 = \{\texttt{a(3:4)}, \texttt{part(2)}, \texttt{part\_copy(2)}, \texttt{m(2)}\}$:

```
arb
    seq
        part(1) = max(a(1), a(2))
        barrier ; part_copy(1) = part(2) ; barrier
        m(1) = max(part(1), part_copy(1))
    end seq
    seq
        part(2) = max(a(3), a(4))
        barrier ; part_copy(2) = part(1) ; barrier
```

```
        m(2) = max(part_copy(2), part(2))
    end seq
end arb
```

which is in turn refined by the following message-passing program P'':

```
arb
    seq
        part(1) = max(a(1), a(2))
        send ("integer", part(1)) to (P2)
        recv (type, part_copy(1)) from (P2)
        m(1) = max(part(1), part_copy(1))
    end seq
    seq
        part(2) = max(a(3), a(4))
        send ("integer", part(2)) to (P1)
        recv (type, part_copy(2)) from (P1)
        m(2) = max(part(2), part_copy(2))
    end seq
end arb
```

5.4 Executing subset-par-model programs

5.4.1 Transformations to practical languages/libraries

We can use the transformation of the preceding section to transform programs in the subset **par** model into programs in any language that supports (1) multiple-address-space parallel composition with (2) single-sender–single-receiver message-passing. Examples include Fortran M [40] (which supports multiple-address-space parallel composition via process blocks and single-sender–single-receiver message-passing via channels) and MPI [58] (which assumes execution in an environment of multiple-address-space parallel composition and supports single-sender–single-receiver message-passing via tagged point-to-point sends and receives).

5.4.2 Example

Program P'' from Section 5.3.2 can be implemented by the following Fortran M program:

```
program main
    integer a(4)
```

```
    inport (integer) inp(2)
    outport (integer) outp(2)
    channel (outp(1), inp(2))
    channel (outp(2), inp(1))
    processes
        process call P(a(1:2), inp(1), outp(1))
        process call P(a(3:4), inp(2), outp(2))
    end processes
end

process P(a, inp, outp)
    integer a(2)
    inport (integer) inp
    outport (integer) outp
    integer part, part_copy, m
    part = max(a(1), a(2))
    send (outp) part
    receive (inp) part_copy
    m = max(part, part_copy)
end process
```

Chapter 6

Extended examples

This chapter presents several examples of programs using **arb** composition and shows how to transform some of them into programs suitable for shared-memory and distributed-memory architectures using the transformations presented in Chapter 3. Observe that in these examples, for simplicity, we emphasize program readability over efficiency; the programs we derive for shared-memory and distributed-memory architectures are more efficient than the **arb**-model programs from which they are produced, but additional transformations could be applied to further improve efficiency.

6.1 2-dimensional FFT

6.1.1 Problem description

This program performs a 2-dimensional FFT in place, as described in [62]. Performing the 2-dimensional FFT on an N by M array is accomplished by performing a 1-dimensional FFT on each row of a 2-dimensional array and then performing a 1-dimensional FFT on each column of the resulting 2-dimensional array.

6.1.2 Program

Clearly the 1-dimensional FFTs on the rows of the array are independent, as are the 1-dimensional FFTs on the columns of the array. We can thus express the desired computation as in Figure 6.1.

6.1.3 Applying our transformations

Program for shared memory. We can apply Theorem 3.2 to produce the program shown in Figure 6.2, which is readily transformed into a program in the **par** model using the transformations of Chapter 4.

```
    integer :: N, M
    complex :: a(N, M)

!---do row FFTs
    arball (i = 1:N)
        call rowfft(a(i, :))
    end arball
!---do column FFTs
    arball (j = 1:M)
        call colfft(a(:, j))
    end arball
```

Figure 6.1: Program for 2-dimensional FFT.

```
    integer :: N, M, P
    complex :: a(N, M)
    integer :: i(P), j(P)

!---do row FFTs
    arball (ip = 1:P)
        do i(ip) = (ip-1)*(N/P) + 1, ip*(N/P)
            call rowfft(a(i(ip), :))
        end do
    end arball
!---do column FFTs
    arball (ip = 1:P)
        do j(ip) = (ip-1)*(M/P) + 1, ip*(M/P)
            call colfft(a(:, j(ip)))
        end do
    end arball
```

Figure 6.2: Program for 2-dimensional FFT, shared-memory version.

Program for distributed memory. To produce a program suitable for distributed memory, we create two copies of array `a`: `a_row` corresponding to a row distribution (convenient for the row FFTs) and `a_col` corresponding to a column distribution (convenient for the column FFTs), as described in Section 3.3.5.4. We map original variable `a` one-to-one onto `a_row` and one-to-one onto `a_col`. If we assume for simplicity that `P` is 2, the resulting **arb**-model program, which is readily transformed into a program in the subset **par** model using the transformations of Chapter 4, is shown in Figure 6.3. For compactness, the two redistribution operations, in which data is copied between `a_row` and `a_col`, are written as Fortran 90 array operations, but observe that each array operation could be expressed as an **arb** composition.

Optimizations. We could reduce the storage requirement of the distributed-memory program by performing the redistribution operation in place (i.e., by aliasing `a_row` and `a_col` and performing the redistribution operations as synchronized multiple assignments rather than as **arb** compositions). Such optimizations are beyond the scope of this thesis, though not, we believe, beyond the scope of

```
    integer :: N, M
    complex :: a_row(N/2, M, 2)
    complex :: a_col(N, M/2, 2)
    integer ::  i(2), j(2)

!---do row FFTs
    arball (ip = 1:2)
        do i(ip) = 1, N/2
            call rowfft(a_row(i(ip), :, ip))
        end do
    end arball
!---redistribute (row to column)
    arb
        a_col(1:(N/2), :, 1) = a_row(:, 1:(M/2), 1)
        a_col(((N/2)+1):N, :, 1) = a_row(:, 1:(M/2), 2)
        a_col(1:(N/2), :, 2) = a_row(:, ((M/2)+1):M, 1)
        a_col(((N/2)+1):N, :, 2) = a_row(:, ((M/2)+1):M, 2)
    end arb
!---do column FFTs
    arball (ip = 1:2)
        do j(ip) = 1, M/2
            call colfft(a_col(:, j(ip), ip))
        end do
    end arball
!---redistribute (column to row)
    arb
        a_row(:, 1:(M/2), 1) = a_col(1:(N/2), :, 1)
        a_row(:, 1:(M/2), 2) = a_col(((N/2)+1):N, :, 1)
        a_row(:, ((M/2)+1):M, 1) = a_col(1:(N/2), :, 2)
        a_row(:, ((M/2)+1):M, 2) = a_col(((N/2)+1):N, :, 2)
    end arb
```

Figure 6.3: Program for 2-dimensional FFT, distributed-memory version.

our models and methodology.

6.2 1-dimensional heat equation solver

6.2.1 Problem description

In this example, the goal is to solve the 1-dimensional heat diffusion equation

$$\frac{\partial U}{\partial t} = \frac{\partial^2 U}{\partial x^2}$$

with boundary condition

$$U(x,t) = 1 \quad \text{for boundary } x \ .$$

Following the method described in [47], we discretize the problem domain (representing the x dimension as an array of N points) and use the following approximation:

$$\frac{U(x_i,t_{k+1}) - U(x_i,t_k)}{\Delta t} = \frac{U(x_{i+1},t_k) - 2U(x_i,t_k) + U(x_{i-1},t_k)}{\Delta x^2}$$

We assume an initial value of 0 for all non-boundary points. The program is to print out values for each point and each timestep.

6.2.2 Program

We are computing a sequence of values for each point in the 1-dimensional array. However, since we can print each out as it is computed, we need only retain two values for each point. Thus, we represent U by two arrays, one for the current time step (`uk`) and one for the next time step (`ukp1`). Clearly the computation of a new value for each element of `ukp1` is independent of the computation of new values for the other elements, and the same is true for the copying of values from `ukp1` to `uk`. Initialization of elements of `ukp1` is also independent. We can thus express the desired computation as in Figure 6.4.

6.2.3 Applying our transformations

Program for shared memory. We first apply Theorem 3.2 to produce the program shown in Figure 6.5, which is readily transformed into a program in the **par** model using the transformations of Chapter 4.

Program for distributed memory. To produce a program suitable for distributed memory, we proceed as in the examples of Section 3.3.5, duplicating constants `dt` and `dx`, and partitioning `uk` and `ukp1` into local sections, with each local section of `uk` surrounded by a ghost boundary of width 1. The resulting **arb**-model program, which is readily transformed into a program in the subset **par** model using the transformations of Chapter 4, is shown in Figure 6.6.

```
    integer :: NX, NSTEPS
    real :: uk(NX), ukp1(NX), dx, dt

    dx = 1.0/NX ; dt = 0.5*dx*dx
!---initialize grid
    arb
        arball (i = 2:NX-1)
            uk(i) = 0.0
        end arball
        uk(1) = 1.0
        uk(NX) = 1.0
    end arb
!---time step loop
    do k = 1, NSTEPS
    !---compute values for next time step
        arball (i = 2:NX-1)
            ukp1(i) = uk(i) + (dt/(dx*dx)) * (uk(i-1) - 2*uk(i) + uk(i+1))
        end arball
    !---save just-computed values for next time step and print
        arball (i = 2:NX-1)
            uk(i) = ukp1(i)
        end arball
        call print_heat(k, uk)
    end do
```

Figure 6.4: Program for 1-dimensional heat equation.

Optimizations. We could eliminate the copying of `ukp1` to `uk` by alternately regarding `uk` and `ukp1` as the "current" values and alternately computing `ukp1` based on the values of `uk` and `uk` based on the values of `ukp1`. Such an optimization could be performed on the original **arb**-model program (as a sequential transformation) and then carried through the transformations for shared and distributed memory.

```
    integer :: NX, NSTEPS, P
    real :: uk(NX), ukp1(NX), dx, dt
    integer :: i(P), ifirst(P), ilast(P)

    dx = 1.0/NX ; dt = 0.5*dx*dx
!---determine first and last interior points for each process
    arball (ip = 1:P)
        ifirst(ip) = max(2, (ip-1)*(NX/P)+1)
        ilast(ip) = min(NX-1, (ip)*(NX/P))
    end arball
!---initialize
    arb
        uk(1) = 1.0
        uk(NX) = 1.0
        arball (ip = 1:P)
            uk(ifirst(ip):ilast(ip)) = 0.0
        end arball
    end arb
!---time step loop
    do k = 1, NSTEPS
    !---compute values for next time step
        arball (ip = 1:P)
            do i(ip) = ifirst(ip), ilast(ip)
                ukp1(i(ip)) = uk(i(ip)) + (dt/(dx*dx)) &
                    * (uk(i(ip)-1) - 2*uk(i(ip)) + uk(i(ip)+1))
            end do
        end arball
    !---save just-computed values for next time step and print
        arball (ip = 1:P)
            uk(ifirst(ip):ilast(ip)) = ukp1(ifirst(ip):ilast(ip))
        end arball
        call print_heat(k, uk)
    end do
```

Figure 6.5: Program for 1-dimensional heat equation, shared-memory version.

```
    integer :: NX, NSTEPS, P
    real :: uk(0:(NX/P)+1, P), ukp1(NX/P, P), dx(P), dt(P)
    integer :: i(P), ifirst(P), ilast(P)

    arball (ip = 1:P)
        dx(ip) = 1.0/NX ; dt(ip) = 0.5*dx(ip)*dx(ip)
    end arball
!---determine first and last interior points for each process
    arb
        ifirst(1) = 2
        arball (ip = 2:P)
            ifirst(ip) = 1
        end arball
    end arb
    arb
        arball (ip = 1:(P-1))
            ilast(ip) = NX/P
        end arball
        ilast(P) = (NX/P)-1
    end arball
!---initialize grid
    arb
        uk(1, 1) = 1.0
        uk(NX/P, P) = 1.0
        arball (ip = 1:P)
            uk(ifirst(ip):ilast(ip), ip) = 0.0
        end arball
    end arb
!---time step loop
    do k = 1, NSTEPS
    !---re-establish copy consistency (boundary exchange)
        arball (ip = 2:P)
            uk(0, ip) = uk(NX/P, ip-1)
        end arball
        arball (ip = 1:(P-1))
            uk((NX/P)+1, ip) = uk(1, ip+1)
        end arball
    !---compute values for next time step
        arball (ip = 1:P)
            do i(ip) = ifirst(ip), ilast(ip)
                ukp1(i(ip), ip) = uk(i(ip), ip) + (dt(ip)/(dx(ip)*dx(ip))) &
                    * (uk(i(ip)-1, ip) - 2*uk(i(ip), ip) + uk(i(ip)+1, ip))
            end do
        end arball
    !---save just-computed values for next time step and print
        arball (ip = 1:P)
            uk(ifirst(ip):ilast(ip), ip) = ukp1(ifirst(ip):ilast(ip), ip)
        end arball
        call print_heat(k, uk)
    end do
```

Figure 6.6: Program for 1-dimensional heat equation, distributed-memory version.

6.3 2-dimensional iterative Poisson solver

This problem is similar in many respects to the heat-equation problem; it is included to show how iteration until convergence differs from fixed iteration.

6.3.1 Problem description

This example is largely based on the discussion of the Poisson problem in [74].[1] The program finds a numerical solution to the Poisson problem

$$-\frac{\partial^2 U}{\partial x^2} - \frac{\partial^2 U}{\partial y^2} = f(x,y)$$

with Dirichlet boundary condition

$$u(x,y) = g(x,y)$$

where f and g are given. The method is to discretize the problem domain, representing U as a 2-dimensional grid of points with spacing h, and use Jacobi iteration, that is, apply the following operation to all interior points until convergence is reached:

$$4u_{(i,j)}^{(k+1)} = h^2 f_{i,j} + u_{(i-1,j)}^{(k)} + u_{(i+1,j)}^{(k)} + u_{(i,j-1)}^{(k)} + u_{(i,j+1)}^{(k)} \quad .$$

The program is to print out only the final (converged) values. For simplicity, we assume that the computation will converge.

6.3.2 Program

We are computing a sequence of values for each point in the 2-dimensional array. However, we need only retain two values for each point, so we represent U by two arrays, one for the current iteration (`uk`) and one for the next iteration (`ukp1`).[2] As in the previous problem, the computation of a new value for each element of `ukp1` is independent of the computation of new values for the other elements, and the same is true for the copying of values from `ukp1` to `uk`. Initialization of elements of `ukp1` is also independent. We can thus express the desired computation as in Figure 6.7. Note the nesting of **arb** and **arball** in the initialization.

[1] We derive a slightly different program because of our focus on readability over efficiency. Note however that nothing in our methodology precludes developing the same program presented in [74].

[2] Actually, we can reduce the storage requirements of the program by reducing the number of points for which we maintain both "current" and "next" values, as in [74]. As noted, however, in these examples we stress readability over efficiency and defer such optimizations.

```
    integer :: NX, NY
    real :: H, TOLERANCE
    external F, G
    real :: uk(NX, NY), ukp1(NX, NY), fvals(NX, NY), diffmax

!---initialize
    arb
        arball (i = 2:NX-1, j = 2:NY-1)
            uk(i,j) = F(i,j,H)
        end arball
        arball (j = 1:NY)
            uk(1,j) = G(1,j,H)
        end arball
        arball (j = 1:NY)
            uk(NX,j) = G(NX,j,H)
        end arball
        arball (i = 2:NX-1)
            uk(i,1) = G(i,1,H)
        end arball
        arball (i = 2:NX-1)
            uk(i,NY) = G(i,NY,H)
        end arball
    end arb
    arball (i = 1:NX, j = 1:NY)
        fvals(i,j) = F(i,j,H)
    end arball
!---compute until convergence
    diffmax = TOLERANCE + 1.0
    do while (diffmax > TOLERANCE)
    !---compute new values
        arball (i = 2:NX-1, j = 2:NY-1)
            ukp1(i,j) = 0.25*(H*H*fvals(i,j)    &
                + uk(i,j-1) + uk(i,j+1)     &
                + uk(i-1,j) + uk(i+1,j))
        end arball
    !---check for convergence:
    !   compute max(abs(ukp1(i,j) - uk(i,j)))
        diffmax = 0.0
        do i = 2, NX-1
        do j = 2, NY-1
            diffmax = max(diffmax, abs(ukp1(i,j) - uk(i,j)))
        end do
        end do
    !---copy new values to old values
        arball (i = 2:NX-1, j = 2:NY-1)
            uk(i,j) = ukp1(i,j)
        end arball
    end do ! while
```

Figure 6.7: Program for 2-dimensional iterative Poisson solver.

6.4 Quicksort

6.4.1 Problem description

The problem is to sort an array of integers. Two variants of the quicksort algorithm are presented: a standard recursive version, and a "one-deep" nonrecursive version more suitable for scalable parallel implementations [26, 68].

6.4.2 Recursive program

This program sorts the array of integers in place. Once the array has been split into two parts, they can be sorted independently, so we can thus express the desired computation as in Figure 6.8.

```
    integer :: N
    integer :: a(N)

    call quicksort_r(a)

!---recursive program definition

    recursive subroutine quicksort_r(a)
    integer, dimension(:), intent(inout) :: a
    integer :: splitpoint

!---if not base case
    if (size(a) > 1) then
    !---partition
        call split(a, splitpoint)
    !---recursively quicksort partitions
        arb
            call quicksort_r(a(1:splitpoint-1))
            call quicksort_r(a(splitpoint+1:size(a)))
        end arb
    end if

    end subroutine quicksort_r
```

Figure 6.8: Recursive quicksort program.

6.4.3 "One-deep" program

This program sorts the input array to produce the output array. In this variant, the array is split into `k` parts, which can then be sorted independently using an in-place sequential sort program `qsort`. We can thus express the desired computation as in Figure 6.9.

```
    integer :: N, K
    integer :: a_in(N), a_out(N)
    integer :: splitpoints(K+1)

!---partition
    call split(a_in, a_out, splitpoints)
!---sort partitions
    arball (i = 1:K)
        call qsort(a_out(splitpoints(i):splitpoints(i+1)-1))
    end arball
```

Figure 6.9: One-deep quicksort program.

Chapter 7

Archetypes for scientific computing

The preceding chapters describe a programming model and methodology that is general and formal. This chapter presents experimental work in support of the archetypes/patterns aspects of the model and methodology; it also presents a more application-oriented view of the methodology, combining the ideas from the preceding chapters with the idea of design patterns.

Hypothesis: An archetypes-related approach to developing parallel applications, as described in Section 7.1, facilitates the writing of correct and efficient application programs by reducing application development to a process of "filling in the blanks" of the archetype-defined framework with essentially sequential code in which interprocess interaction is limited to encapsulated archetype-defined data-exchange operations.

Experiment: We identify example archetypes (Section 7.2), develop for each archetype an "implementation" (code framework/library and documentation), and use these implementations to develop applications (Section 7.3), some based on existing sequential applications. We consider whether the resulting applications do indeed have the form we describe — essentially sequential code, with interprocess interaction achieved via encapsulated data-exchange operations — and whether they are correct and efficient. Objective evaluation of claims about ease of use is difficult, but we consider whether there is reason to believe that it is easier to write a program using an archetype than to write the same program without an archetype.

Conclusions: The applications we developed have the desired form (as shown by the examples in Section 7.3.1) and are correct. In general they are acceptably efficient (as shown in Section 7.3), with the exceptions limited by a poor computation-to-communication ratio and possibly by untuned archetype implementations (and tuning the archetype implementations — i.e., libraries — should improve performance for all applications). These results support our claim that the archetypes approach is a good one.

7.1 Parallel program archetypes

A great deal of work has been done on methods of exploiting design patterns in program development. The work described in this chapter restricts attention to one kind of pattern that is relevant in parallel programming: the pattern of the parallel computation and communication structure.

Methods of exploiting design patterns in program development begin by identifying classes of problems with similar computational structures and creating abstractions that capture the commonality. Combining a problem class's computational structure with a parallelization strategy gives rise to a dataflow pattern and hence a communication structure. It is this combination of computational structure, parallelization strategy, and the implied pattern of dataflow and communication that we capture as a *parallel program archetype*, or just an *archetype*.

In terms of the programming model presented in previous chapters, the commonality captured by the archetype abstraction makes it possible to develop semantics-preserving transformations applicable to programs that fit the archetype. In particular, the common dataflow pattern makes it possible to encapsulate those parts of the computation that involve interprocess communication and transform them only once, with the results of the transformation made available as a *communication operation* usable in any program that fits the archetype.

7.1.1 Archetype-based assistance for application development

Although the dataflow pattern is the most significant aspect of an archetype in terms of its usefulness in easing the task of developing parallel programs, including computational structure as part of the archetype abstraction helps in identifying the dataflow pattern and also provides some of the other benefits associated with patterns. Such archetypes are useful in many ways:

Program skeletons and code libraries. A program skeleton and code library can be created for each archetype, where the skeleton deals with process creation and interaction between processes, and the code library encapsulates details of the interprocess interaction. If a sequential program fits an archetype, then a parallel program can be developed by fleshing out the skeleton, making use of the code library. The fleshing-out steps deal with defining the *sequential* structure of the processes. Thus, programmers can focus their attention primarily on sequential programming issues.

Amortization of performance optimization costs. One way to achieve portability and performance is to implement common patterns of parallel structures — those for a particular archetype or archetypes — on different target architectures (e.g., multicomputers, symmetric multiprocessors, and non-uniform-memory-access multiprocessors), tuning the implementation to obtain good performance. The cost of this performance optimization effort is amortized over

all programs that fit the pattern.

Assistance with parallelization. Programmers often develop parallel applications by transforming sequential programs. The process of transformation can be laborious and error-prone. However, this transformation process can be systematized for sequential programs that fit specific computational patterns; then, if a sequential program fits one of these patterns (archetypes), the transformation steps appropriate to that pattern can be used.

In some cases, parallelizing compilers can generate programs that execute more efficiently on parallel machines if programmers provide information about their programs in addition to the program text itself. Although the focus of this part of our work is on active stepwise refinement by programmers and not on compilation tools, we postulate that the dataflow pattern is information that can be exploited by a compiler.

Framework for program design, development, and reasoning. Just as the identification of computational patterns in object-oriented design is useful in teaching systematic sequential program design, identification of computational and dataflow patterns (archetypes) is helpful in teaching parallel programming. Similarly, just as the use of computational patterns can make reasoning about sequential programs easier by providing a framework for proofs of algorithmic correctness, archetypes can provide a framework for reasoning about the correctness of parallel programs. Archetypes can also provide frameworks for testing and documentation.

Performance models. Archetypes may also be helpful in developing performance models for classes of programs with common structure, as discussed in [64].

Program composition. Archetypes can be useful in structuring programs that combine task and data parallelism, as described in [23].

7.1.2 An archetype-based program development strategy

Our general strategy for writing programs using archetypes is as follows:

Start with a sequential algorithm or problem description.

Identify an appropriate archetype.

Develop an initial archetype-based version of the algorithm. This initial version is structured according to the archetype's pattern and gives an indication of the concurrency to be exploited by the archetype. (In terms of our programming model, this initial version is a program in the **arb** model.) Essentially, this step consists of structuring the original algorithm to fit the archetype pattern and "filling in the blanks" of the archetype with application-specific

details. Transforming the original algorithm into this archetype-based equivalent can be done in one stage or via a sequence of smaller transformations; in either case, it is guided by the archetype pattern.

An important feature of this initial archetype-based version of the algorithm is that it can be executed sequentially (by converting **arb** composition or its equivalent into sequential composition, as described in the examples). For deterministic programs, this sequential execution gives the same results as parallel execution; this allows debugging in the sequential domain using familiar tools and techniques.

Transform the initial archetype-based version of the algorithm into an equivalent algorithm suitable for efficient execution on the target architecture. (In terms of our programming model, the objective in this step is to produce a program in the **par** or the subset **par** model, depending on the target architecture.) The archetype assists in this transformation, either via guidelines to be applied manually or via automated tools. Again, the transformation can optionally be broken down into a sequence of smaller stages, and in some cases intermediate stages can be executed (and debugged) sequentially. A key aspect of this transformation process is that the transformations defined by the archetype preserve semantics and hence correctness. During this transformation process, portions of the program that correspond to archetype-defined communication operations can be replaced with calls to archetype-defined library routines.

Implement the efficient archetype-based version of the algorithm using a language or library suitable for the target architecture. Here again the archetype assists in this process, not only by providing suitable transformations (either manual or automatic), but also by providing program skeletons and/or libraries that encapsulate some of the details of the parallel code (process creation, message-passing, and so forth).

A significant aspect of this step is that it is only here that the application developer must choose a particular language or library; the algorithm versions produced in the preceding steps can be expressed in any convenient notation, since the ideas are essentially language-independent.

This chapter presents example archetypes and shows how they and this strategy can be used to develop applications. Our work to date has concentrated on target architectures with distributed memory and message-passing, and the discussion reflects this focus, but we believe that the work has applicability for shared-memory architectures as well, particularly those with local caches and/or explicit cache control.

7.2 Example archetypes

As discussed earlier, in order to test our hypothesis about the value of an archetypes-based approach to developing parallel applications, we identified some example archetypes, developed archetype implementations consisting of code libraries and documentation, and used these implementations to develop application programs. This section presents some example archetypes developed as the first phase of this experimental work.

7.2.1 The mesh-spectral archetype

7.2.1.1 Computational pattern

A number of scientific computations can be expressed in terms of operations on N-dimensional grids. While it is possible to abstract from such computations patterns resembling higher-order functions (like that of traditional divide and conquer, for example), our experience with complex applications suggests that such patterns tend to be too restrictive to address complex problems. Instead, the pattern captured by the mesh-spectral archetype[1] is one in which the overall computation is based on N-dimensional grids (where N is usually 1, 2, or 3) and structured as a sequence of the following operations on those grids:

Grid operations, which apply the same operation to each point in the grid, using data for that point and possibly neighboring points. If the operation uses data from neighboring points, the set of variables modified in the operation must be disjoint from the set of variables used as input. Input variables may also include "global" variables (variables common to all points in the grid, e.g., constants). In terms of our model, a grid operation can be expressed as an **arball** over all or most points in the grid.

Row (column) operations, which apply the same operation to each row (column) in the grid. Analogous operations can be defined on subsets of grids with more than 2 dimensions. The operation must be such that all rows (columns) are operated on independently — that is, the calculation for row i cannot depend on the results of the calculation for row j, where $i \neq j$. In terms of our model, a row (column) operation can be expressed as an **arball** over all rows (columns) in the grid.

Reduction operations, which combine all values in a grid into a single value (e.g., finding the maximum element). In terms of our model, reduction operations can be transformed into computations with exploitable concurrency (**arb** or **arball** composition) by using the transformations described in Section 3.4.1.

[1] We call this archetype "mesh-spectral" because it combines and generalizes two other archetypes, a mesh archetype focusing on grid operations, described in Section 7.2.3, and a spectral-methods archetype focusing on row and column operations, described in Section 7.2.2.

File input/output operations, which read or write values for a grid. In terms of our model, file input/output operations can be transformed into computations with exploitable concurrency (**arb** or **arball** composition) if it is possible to replace the underlying sequential file operations with file operations that allow for concurrency.

Data may also include global variables common to all points in the grid (constants, for example, or the results of reduction operations), and the computation may include simple control structures based on these global variables (for example, looping based on a variable whose value is the result of a reduction).

7.2.1.2 Parallelization strategy and dataflow

In terms of our model, "parallelizing" a program means developing an equivalent program in the **par** model or the subset **par** model — usually the latter, since it corresponds to the architectures in which we are most interested, those in which memory is distributed. Devising a parallelization strategy for a particular archetype begins by considering how its dataflow pattern can be used to determine how to distribute data among processes in such a way that communication requirements are minimized.

For the mesh-spectral archetype, the dataflow patterns of the archetype's characteristic operations lend themselves to a data distribution scheme based on partitioning the data grid into regular contiguous subgrids (local sections) and distributing them among processes. As described in this section, some operations impose requirements on how the data is distributed, while others do not.

Grid operations. Provided that the restriction in Section 7.2.1.1 is met, points can be operated on in any order or simultaneously. Thus, each process can compute (sequentially) values for the points in its local section of the grid, and all processes can operate concurrently. Grid operations impose no restrictions on data distribution, although the choice of data distribution may affect the resulting program's efficiency.[2]

Row (column) operations. Provided that the restriction in Section 7.2.1.1 is met, rows can be operated on simultaneously or in any order. These operations impose restrictions on data distribution: Row operations require that data be distributed by rows, while column operations require that data be distributed by columns.

Reduction operations. Provided that the operation used to perform the reduction is associative (e.g., maximum) or can be so treated (e.g., floating-point addition, if some degree of nondeterminism is acceptable), reductions can be computed concurrently by allowing each process

[2]This chapter addresses only the question of which data distributions are compatible with the problem's computational structure. Within these constraints, programmers may choose any data distribution; choosing the data distribution that gives the best performance is important but orthogonal to the concerns of this chapter. However, an archetype-based performance model, such as that described in [64], may help with this choice.

to compute a local reduction result and then combining them, for example via recursive doubling. Reduction operations, like grid operations, may be performed on data distributed in any convenient fashion. After completion of a reduction operation, all processes have access to its result.

File input/output operations. Exploitable concurrency and appropriate data distribution depend on considerations of file structure and (perhaps) system-dependent I/O considerations. One possibility is to operate on all data sequentially in a single process, which implies a data "distribution" in which all data is collected in a single process. Another possibility is to perform I/O "concurrently" in all processes (actual concurrency may be limited by system or file constraints), using any convenient data distribution.

Patterns of communication (in distributed-memory versions of mesh-spectral applications) arise as a consequence of how these operations are composed to form an individual algorithm; if two operations requiring different data distributions are composed in sequence, they must be separated by data redistribution. Distributed memory introduces the additional requirement that each process have a duplicate copy of any global variables, with their values kept synchronized — that is, any change to such a variable must be duplicated in each process before the value of the variable is used again. A key element of this archetype is support for ensuring that these requirements are met. This support can take the form of guidelines for manually transforming programs, as in our archetype-implementation user guides [34, 57, 33], or it could be expressed in terms of more formal transformations with arguments for their correctness, as in the transformations of Chapter 3.

7.2.1.3 Communication patterns

This data-distribution scheme thus gives rise to the need (in distributed memory) for a small set of communication operations:

Grid redistribution. If different parts of the computation require different distributions — for example, if a row operation is followed by a column operation — data must be redistributed among processes, as in Figure 7.1.

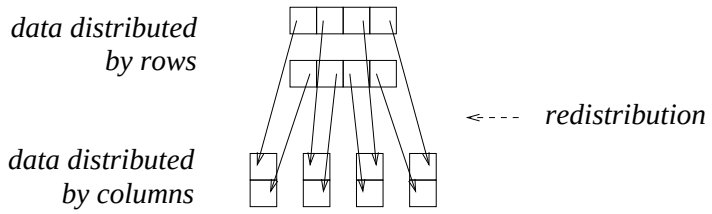


Figure 7.1: Redistribution: rows to columns.

Exchange of boundary values. If a grid operation uses value from neighboring points, points on the boundary of each local section requires data from neighboring processes' local sections. This dataflow requirement can be met by surrounding each local section with a *ghost boundary* containing shadow copies of boundary values from neighboring processes and using a boundary-exchange operation (in which neighboring processes exchange boundary values) to refresh these shadow copies, as shown in Figure 7.2.

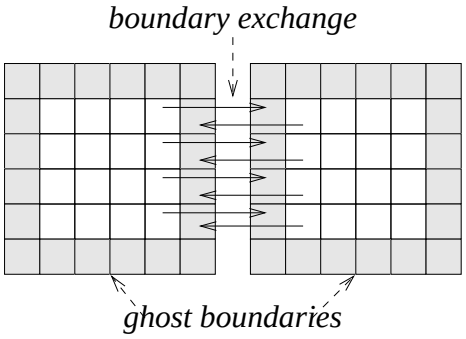


Figure 7.2: Boundary exchange.

Broadcast of global data. When global data is computed (or changed) in one process only (for example, if it is read from a file), a broadcast operation is required to re-establish copy consistency.

Support for reduction operations. Reduction operations can be supported by several communication patterns depending on their implementation — for example, all-to-one/one-to-all or recursive doubling. Figure 7.3 shows recursive doubling used to compute the sum of the elements of an array.

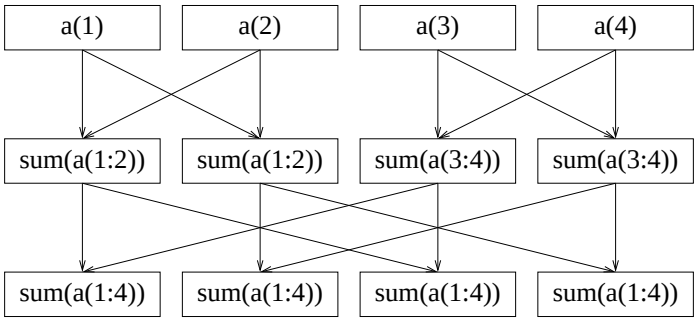


Figure 7.3: Recursive doubling to compute a reduction (sum).

Support for file input/output operations. File input/output operations can be supported by several communication patterns, e.g., data redistribution (one-to-all or all-to-one).

All of the required operations can be supported by a communication library containing a boundary-exchange operation, a general data-redistribution operation, and a general reduction operation. Each operation corresponds to a data-exchange operation, as described in Chapter 5, and hence implies a synchronization across the components of the parallel composition before and after the exchanging of data. It is straightforward to write down specifications of these operations in terms of pre- and postconditions (which is helpful in determining where they should be used); these specifications can then be implemented in any desired language or library as part of an archetype implementation.

7.2.1.4 Implementation

We have developed for the mesh-spectral archetype an implementation consisting of a code skeleton and an archetype-specific library of communication routines for the operations described in Section 7.2.1.3, with versions based on Fortran M [40] and Fortran with MPI [58]. The implementation is described in detail in [34]. The Fortran M version has been used to run applications on the IBM SP and on networks of Sun workstations; the MPI version has been used to run applications on the IBM SP and on networks of Sun and Pentium-based workstations.

7.2.2 The spectral archetype

The spectral archetype is a strict subset of the mesh-spectral archetype (Section 7.2.1), in which the allowed computational operations consist of row and column operations, reduction operations, and file input/output operations. Parallelization strategy, dataflow patterns, and required communication operations are thus a subset of the corresponding entities for the mesh-spectral archetype. The communication operations consist of a restricted form of redistribution — row to column and vice versa — and support for reduction operations and file input/output.

We have developed for this archetype an implementation (based on Fortran M [40]) consisting of a code skeleton and an archetype-specific library of communication routines. The implementation is described in detail in [33]; it has been used to run an application on the IBM SP and on a network of Sun workstations.

7.2.3 The mesh archetype

The mesh archetype is a strict subset of the mesh-spectral archetype (Section 7.2.1), with one minor change. The allowed computational operations consist of grid operations, reduction operations, and file input/output operations. The parallelization strategy, however, is based on a maximum of two data distributions: one in which data is partitioned and distributed among *grid processes* (in the

same manner described for the mesh-spectral archetype) and one in which all data resides in a *host process*. Grid computations and reduction operations may be performed on data distributed in either manner, although obviously they will not be very efficient when performed on data in a host-only distribution. File input/output operations may also be performed on data distributed in either manner, although depending on the details of the target architecture they may be much simpler when performed on data in a host-only distribution. (This is the justification for including a host process in the archetype.) Dataflow patterns are the subset of mesh-spectral dataflow patterns implied by the just-described computational patterns and parallelization strategy. The required communication operations consist of boundary exchange, a restricted form of data redistribution (host to grid and vice versa), and support for reduction operations and file input/output.

We have developed for this archetype an implementation consisting of a code skeleton and an archetype-specific library of communication routines,[3] with versions based on Fortran M [40], Fortran with p4 [17], and Fortran with NX [60]. The implementation is described in detail in [57]; it has been used to run applications on the IBM SP, the Intel Delta, the Intel Paragon, and a network of Sun workstations.

7.3 Applications

This section discusses the second phase of the archetypes-related experimental work, in which we used the example archetypes presented in Section 7.2 to develop applications. The examples in Section 7.3.1 illustrate how the mesh-spectral archetype can be used to develop algorithms and transform them into versions suitable for execution on a distributed-memory–message-passing architecture. In addition, Section 7.3.2 briefly describes more complex applications based on this archetype.

7.3.1 Development examples

This section presents two examples of program development using the mesh-spectral archetype of Section 7.2.1. These examples illustrate that the key benefits of developing an algorithm using the mesh-spectral archetype are (1) the guidelines or transformations for converting the algorithm to a form suitable for the target architecture, and (2) the encapsulated and reusable library of communication operations. The performance of the resulting programs is to a large extent dependent on the performance of this communication library, but our experiences as sketched in this section and the following section suggest that even fairly naive implementations of the communication library can give acceptable performance. Performance can then be improved by tuning the library routines,

[3] As an interesting aside, these communication routines were explicitly developed using the method described in Chapter 5, in which a set of assignment statements is transformed into a set of message-passing commands.

with potential benefit for other archetype-based applications.

7.3.1.1 2-dimensional FFT

We first present a simple example making use of row and column operations and data redistribution. This example illustrates how the archetype guides the process of transforming a sequential algorithm into a program for a distributed-memory–message-passing architecture.

Problem description. The problem is that described in Section 6.1.

Archetype-based algorithm, version 1. It is clear that the sequential algorithm described fits the pattern of the mesh-spectral archetype: The data (the 2-dimensional array) is a grid, and the computation consists of a row operation followed by a column operation. Thus, it is easy to write down an archetype-based version of the algorithm. Figure 7.4 shows HPF[43]-like pseudocode for this version. Observe that since the iterations of each `FORALL` are independent, this algorithm can be executed (and debugged, if necessary) sequentially by replacing each `FORALL` with a `DO` loop. Observe also that this algorithm could be executed without change and with the same results on an architecture that supports the `FORALL` construct. This equivalence of results for parallel and sequential execution is a consequence of the definitions and theorems of Chapter 2.

```
    integer N, M
    complex :: a(N, M)

!---do row FFTs
    !HPF$ INDEPENDENT
    forall (i = 1:N)
        call rowfft(a(i,:))
    end forall
!---do column FFTs
    !HPF$ INDEPENDENT
    forall (j = 1:M)
        call colfft(a(:,j))
    end forall
```

Figure 7.4: Program for 2-dimensional FFT, version 1.

Archetype-based algorithm, version 2. We next consider how to transform the initial version of the algorithm into a version suitable for execution on a distributed-memory–message-passing architecture. For such an architecture, the archetype can be expressed as an SPMD computation with `P` processes, with the archetype supplying any code skeleton needed to create and connect the `P` processes. Guided by the archetype (i.e., by the discussion of dataflow and communication patterns in Section 7.2.1), we can transform the algorithm of Figure 7.4 into an SPMD computation in which

each process executes the pseudocode shown in Figure 7.5: Since the precondition of the row operation is that the data be distributed by rows, and the precondition of the column operation is that the data be distributed by columns, we must insert between these two operations a data redistribution. For the sake of tidiness, we add an additional data redistribution after the column operation to restore the initial data distribution. Observe that most of the details of interprocess communication are encapsulated in the redistribution operation, which can be provided by an archetype-specific library of communication routines, freeing the application developer to focus on application-specific aspects of the program.

```
    integer :: N, M, P
    complex :: a_rows(N/P, M)
    complex :: a_cols(N, M/P)

!---do row FFTs
    do i = 1, N/P
        call rowfft(a_rows(i,:))
    end do
!---redistribute
    call redistribute(a_rows, a_cols)
!---do column FFTs
    do j = 1, M/P
        call colfft(a_cols(:,j))
    end do
!---redistribute to restore original distribution
    call redistribute(a_cols, a_rows)
```

Figure 7.5: Program for 2-dimensional FFT, version 2.

Implementation. Transformation of the algorithm shown in Figure 7.5 into code in a sequential language plus message-passing is straightforward, with most of the details encapsulated in the redistribution routine. This algorithm has been implemented using the mesh-spectral archetype implementation described in Section 7.2.1. Figure 7.6 shows execution times and speedups for the MPI version of the parallel code, executing on an IBM SP. Speedups are relative to the equivalent sequential code (produced by executing version 1 of the algorithm sequentially) executed on one processor. Disappointing performance is a result of too small a ratio of computation to communication. This parallelization of a 2-dimensional FFT might nevertheless be sensible as part of a larger computation or for problems exceeding the memory requirements of a single processor.

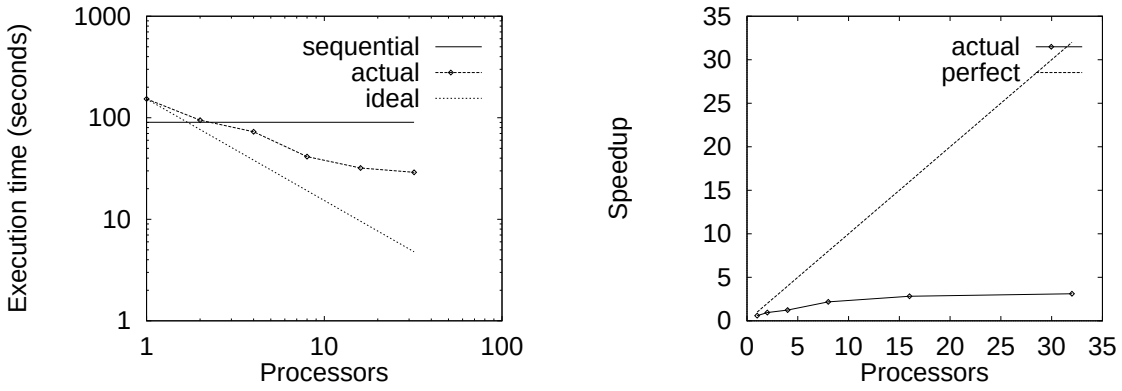


Figure 7.6: Execution times and speedups for parallel 2-dimensional FFT compared to sequential 2-dimensional FFT for 800 by 800 grid, FFT repeated 10 times, using Fortran with MPI on the IBM SP.

7.3.1.2 Poisson solver

We next present a less simple example making use of grid operations, a reduction operation, and the use of a global variable for control flow. This example again illustrates how the archetype guides the process of transforming a sequential algorithm into a program for a distributed-memory–message-passing architecture.

Problem description. The problem is that described in Section 6.3. Details of the initialization phase of the computation have been omitted in the interest of brevity.

Archetype-based algorithm, version 1. The sequential algorithm described fits the pattern of the mesh-spectral archetype: The data consists of several grids (`uk`, `ukp1`, and `fvals`) and a global variable `diffmax` that is computed as the result of a reduction operation and used in the program's control flow. Thus, it is straightforward to write down an archetype-based version of the algorithm. Figure 7.7 shows HPF-like pseudocode for this version, using a grid with dimensions `NX` by `NY`. Observe that since the iterations of each `FORALL` are independent, this algorithm can be executed (and debugged, if necessary) sequentially by replacing each `FORALL` with nested `DO` loops. Observe also that this algorithm could be executed without change and with the same results on an architecture that supports the `FORALL` construct, since the iterations of the `FORALL` are independent and the reduction operation (a global maximum) is based on an associative operation. This equivalence of results for parallel and sequential execution is a consequence of the definitions and theorems of Chapter 2.

Archetype-based algorithm, version 2. We next consider how to transform the initial version of the algorithm into a version suitable for execution on a distributed-memory–message-passing architecture. As with the 2-dimensional FFT program, the overall computation is to be expressed as an SPMD computation, with the archetype supplying any code skeleton needed to create and connect the processes. Since the operations that make up the computation have no data-distribution requirements, it is sensible to write the program using a generic block distribution (distributing data in contiguous blocks among `NPX*NPY` processes conceptually arranged as an `NPX` by `NPY` grid); we can later adjust the dimensions of this process grid to optimize performance. Guided by the archetype (i.e., by the discussion of dataflow and communication patterns in Section 7.2.1), we can transform the algorithm of Figure 7.7 into an SPMD computation in which each process executes the pseudocode shown in Figure 7.8: The program's grids are distributed among processes, with each local section surrounded by a ghost boundary to contain the data required by the grid operation that computes `ukp1`. The global variable `diffmax` is duplicated in each process; copy consistency is maintained because each copy's value is changed only by operations that establish the same value

```
    integer :: NX, NY
    real :: H, TOLERANCE
    real :: uk(NX, NY), ukp1(NX, NY), fvals(NY, NY)
    real :: diffmax

!---initialize
    initialize_poisson(uk, fvals)
!---compute until convergence
    diffmax = TOLERANCE + 1.0
    do while (diffmax > TOLERANCE)
    !---compute new values
        !HPF$ INDEPENDENT
        forall (i = 2:NX-1, j = 2:NY-1)
            ukp1(i,j) = 0.25*(H*H*fvals(i,j)    &
                + uk(i,j-1) + uk(i,j+1) + uk(i-1,j) + uk(i+1,j))
        end forall
    !---check for convergence:
    !   compute max(abs(ukp1(i,j) - uk(i,j)))
        diffmax = 0.0
        do i = 2, NX-1
        do j = 2, NY-1
            diffmax = max(diffmax, abs(ukp1(i,j) - uk(i,j)))
        end do
        end do
    !---copy new values to old values
        !HPF$ INDEPENDENT
        forall (i = 2:NX-1, j = 2:NY-1)
            uk(i,j) = ukp1(i,j)
        end forall
    end do ! while
```

Figure 7.7: Poisson solver, version 1.

in all processes (initialization and reduction). Each grid operation is distributed among processes, with each process computing new values for the points in its local section. (Observe that new values are computed only for points in the intersection of the local section and the whole grid's interior.) To satisfy the precondition of a grid operation using data from neighboring points, the computation of `ukp1` is preceded by a boundary exchange operation. The reduction operation is also transformed in the manner described previously; since a postcondition of this operation is that all processes have access to the result of the reduction, copy consistency is re-established for loop-control variable `diffmax` before it is used. As with the previous example, all of these transformations can be assisted by the archetype, via any combination of guidelines, formally-verified transformations, or automated tools that archetype developers choose to create. Also as with the previous example, observe that most of the details of interprocess communication are encapsulated in the boundary-exchange and reduction operations, which can be provided by an archetype-specific library of communication routines, freeing the application developer to focus on application-specific aspects of the program.

```
    integer :: NX, NY
    real :: H, TOLERANCE
    integer :: NPX, NPY
    real :: uk(0:(NX/NPX)+1, 0:(NY/NPY)+1)
    real :: ukp1(0:(NX/NPX)+1, 0:(NY/NPY)+1)
    real :: fvals(0:(NX/NPX)+1, 0:(NY/NPY)+1)
    real :: diffmax, local_diffmax
    integer :: ilo, ihi, jlo, jhi

!---initialize
    call initialize_poisson_section(uk, fvals)
!---compute intersection of "interior" with local section
    call xintersect(2,NX-1,ilo,ihi)
    call yintersect(2,NY-1,jlo,jhi)
!---compute until convergence
    diffmax = TOLERANCE + 1.0
    do while (diffmax > TOLERANCE)
    !---compute new values
        call boundary_exchange(uk)
        do j = jlo, jhi
        do i = ilo, ihi
            ukp1(i,j) = 0.25*(H*H*f(i,j)        &
                + uk(i,j-1) + uk(i,j+1) + uk(i-1,j) + uk(i+1,j))
        end do
        end do
    !---check for convergence:
    !   compute max(abs(ukp1(i,j) - uk(i,j)))
        local_diffmax = 0.0
        do j = jlo, jhi
        do i = ilo, ihi
            diffmax = max(diffmax, abs(ukp1(i,j) - uk(i,j)))
        end do
        end do
        diffmax = reduce_max(local_diffmax)
    !---copy new values to old values
        uk(ilo:ihi,jlo:jhi) = ukp1(ilo:ihi,jlo:jhi)
    end do ! while
```

Figure 7.8: Poisson solver, version 2.

Implementation. As in the previous example, transformation of the algorithm shown in Figure 7.8 into code in a sequential language plus message-passing is straightforward, with most of the details encapsulated in the boundary-exchange and reduction routines. This algorithm has been implemented using the mesh-spectral archetype implementation described in Section 7.2.1. Figure 7.9 shows execution times and speedups for the MPI version of the parallel code, executing on the IBM SP. Speedups are relative to the equivalent sequential code executed on one processor.

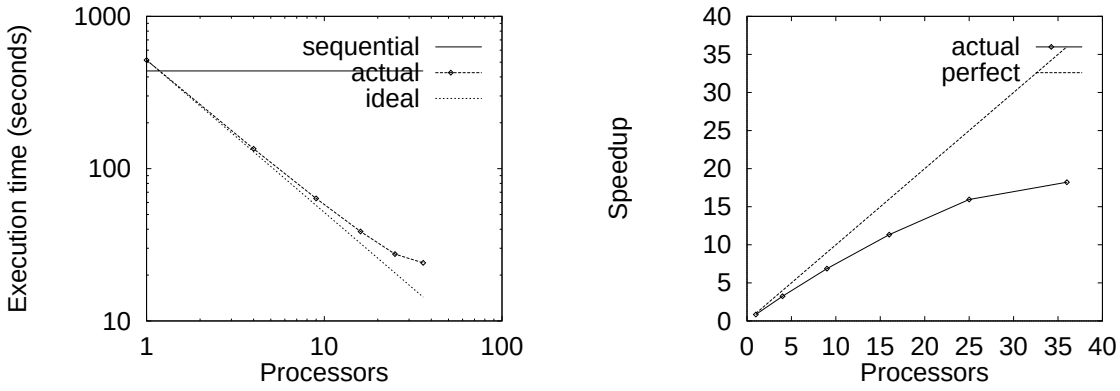


Figure 7.9: Execution times and speedups for parallel Poisson solver compared to sequential Poisson solver for 800 by 800 grid, 1000 steps, using Fortran with MPI on the IBM SP.

7.3.2 Other applications

This section describes additional applications we have developed based on the mesh-spectral archetype and its subsets (the mesh and spectral archetypes).

7.3.2.1 Compressible flow

Two similar computational fluid dynamics codes have been developed using archetypes. These two codes simulate high Mach number compressible flow using a conservative and monotonicity-preserving finite difference scheme [63]. Both are based on the 2-dimensional mesh archetype and have been implemented in Fortran with NX for the Intel Delta and the Intel Paragon. Figure 7.10 shows execution times and speedups for the first code, executing on the Intel Delta. Speedups are relative to single-processor execution of the parallel code. The second version of the code [65] is notable for the fact that it was developed by an "end user" (applied mathematician) using the mesh archetype implementation and documentation, with minimal assistance from the archetype developers.

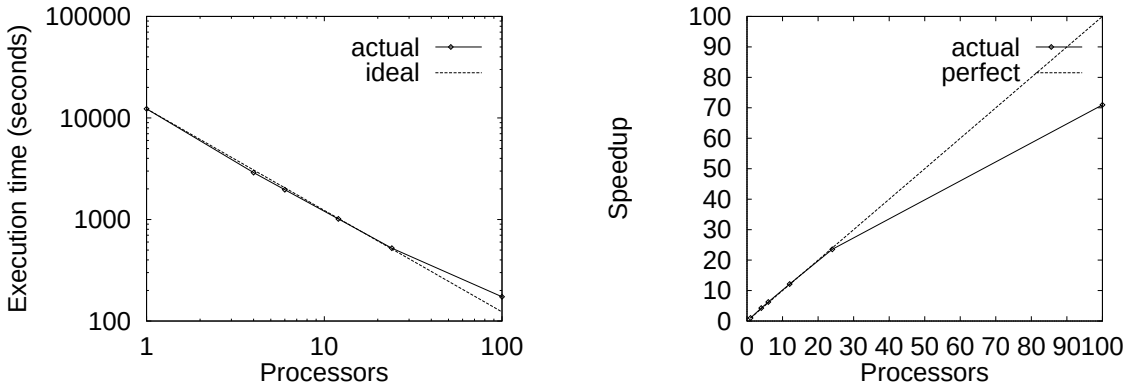


Figure 7.10: Execution times and speedups for 2-dimensional CFD code for 150 by 100 grid, 600 steps, using Fortran with NX on the Intel Delta. Data supplied by Rajit Manohar.

7.3.2.2 Electromagnetic scattering

This code performs numerical simulation of electromagnetic scattering, radiation, and coupling problems using a finite difference time domain technique. It is described further in Chapter 8.

7.3.2.3 Incompressible flow

This spectral code provides a numerical solution of the 3-dimensional Euler equations for incompressible flow with axisymmetry. Periodicity is assumed in the axial direction; the numerical scheme [18] uses a Fourier spectral method in the periodic direction and a fourth-order finite difference

method in the radial direction. It is based on the 2-dimensional spectral archetype and has been implemented in Fortran M for networks of Sun workstations and the IBM SP. Figure 7.11 shows executions and speedups for the parallel code, executing on the IBM SP. Because single-processor execution was not feasible due to memory requirements, a minimum of 5 processors was used, and so speedups are calculated relative to a base of execution on 5 processors. Inefficiencies in executing the code on the base number of processors (e.g., paging) probably explain the better-than-ideal speedup for small numbers of processors.

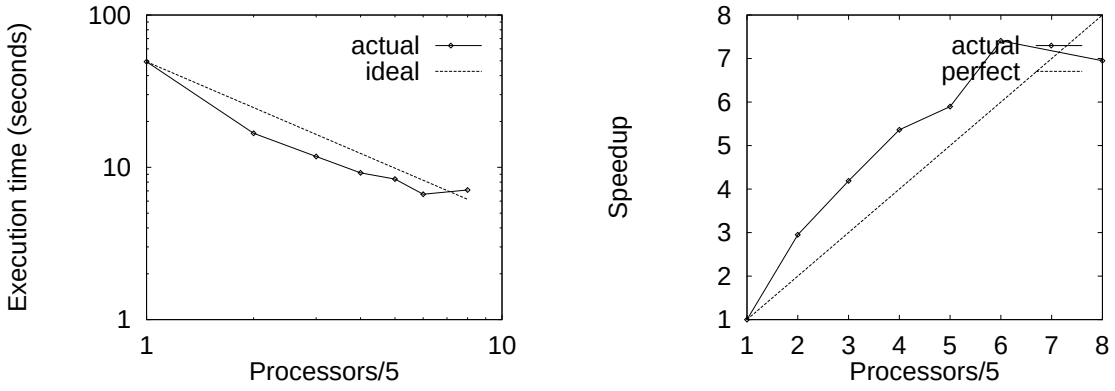


Figure 7.11: Execution times and speedups for spectral code for 1536 by 1024 grid, 20 steps, using Fortran M on the IBM SP. Data supplied by Greg Davis.

7.3.2.4 Airshed model

This code, known as the CIT airshed model [29, 30, 31] models smog in the Los Angeles basin. It is conceptually based on the mesh-spectral archetype, although it does not use the mesh-spectral implementation, and has been implemented on a number of platforms, including the Intel Delta, the Intel Paragon, the Cray T3D, and the IBM SP2, as described in [29].

Chapter 8

Stepwise parallelization methodology

This chapter presents experimental work in support of the transformational aspects of our model and methodology.

Hypothesis: A stepwise archetypes-based approach to parallelizing sequential code, as described in Section 8.1, facilitates the process of parallelizing sequential code by (1) providing a framework for transforming the program, essentially reducing the process to filling in the blanks of a general pattern, as described in Section 7.1, and (2) allowing any needed debugging to be performed in the sequential domain. Ideally the transformations required to produce the simulated-parallel version would also be proved correct, as in Chapter 3, making debugging unnecessary, but in this experiment we provide such proof only for the final transformation from simulated-parallel to parallel, in the belief that it is this transformation the results of which are most difficult to validate by testing and debugging and thus it is this transformation for which a proof is most important.

Experiment: Using one of the archetypes from the preceding experiment (discussed in Chapter 7), we apply the described parallelization methodology to two versions of a sequential application program. We consider whether the resulting application programs have the form we describe above and whether they are correct and efficient. With regard to correctness, we particularly observe whether the final transformation — from "sequential simulated-parallel" to parallel — does in practice as well as in theory preserve semantics, thereby justifying our claim that any needed debugging can be done in the sequential domain, with correctness of the parallel program guaranteed. Objective evaluation of claims about ease of use is difficult, but we consider whether there is reason to believe that it is easier to parallelize a program with our

methodology than without it.

Conclusions: The applications programs we developed have the desired form, insofar as they fit the archetype. The process of parallelization was somewhat cumbersome, but many steps have potential for being automated. As claimed, the final transformation from simulated-parallel to real parallel preserved semantics, with the two programs producing identical results — and in both cases this happened on the first execution of the real parallel program. The resulting parallel programs were reasonably efficient as well. These results support our belief both in the merits of an archetypes-based approach to parallelization (because the parallelism-specific parts of the parallel programs were encapsulated in the archetype-supplied routines) and in the merits of provably semantics-preserving transformations (because the transformation for which a proof was developed did in fact produce correct results with no need for debugging).

8.1 The methodology

In the experiments described in this chapter, we make use of an earlier version of our basic methodology as described in Section 7.1. The overall approach — begin with a sequential program that fits an archetype and apply a sequence of semantics-preserving transformations to produce an equivalent parallel program — is the same, but in this version, the key intermediate stage in the transformation process is what we call the *sequential simulated-parallel version* of the program. This version essentially simulates the operation of a program consisting of N processes executing on a distributed-memory–message-passing architecture and is conceptually very similar to the **arb**-model program behind a subset-**par**-model program. It has the following characteristics:

- The atomic data objects[1] of the program are partitioned into N groups, one for each simulated process; the i-th group simulates the local data for the i-th process. These data objects may include duplicated variables, as discussed in Section 3.3.
- The computation, like the computation of the subset-**par**-model programs of Chapter 5, consists of an alternating sequence of local-computation blocks and data-exchange operations.
- Each local-computation block is a composition of N **arb**-compatible elements, in which the i-th element accesses only local data for the i-th simulated process. Such blocks correspond to sections of the parallel program in which processes execute independently and without interaction.
- Each data-exchange operation consists of an **arb**-compatible set of assignment statements, with the following restrictions:

[1] An atomic data object is as defined in HPF [43]: one that contains no subobjects — e.g., a scalar data object or a scalar element of an array.

- No left-hand or right-hand side may reference atomic data objects belonging to more than one of the N simulated-local-data partitions. The left-hand and right-hand sides of an assignment may, however, reference data from different partitions.
- For each simulated process i, at least one assignment statement must assign a value to a variable in i's local data.

Such blocks correspond to sections of the parallel program in which processes exchange messages: Each assignment statement can be implemented as a single point-to-point message-passing operation, and a group of message-passing operations with a common sender and a common receiver can be combined for efficiency.

Since such a program is a program in our **arb** model, it can be executed sequentially without changing its semantics. Further, it can be transformed in a straightforward and semantics-preserving way into a program in the subset **par** model, or directly into a program in a message-passing notation. We present a semantics-preserving method for the latter transformation in Section 8.2 in this chapter, with a proof of its correctness.

8.2 Supporting theory

In this section we present a general theorem allowing us to transform sequential simulated-parallel programs into equivalent parallel programs. The underlying concepts are similar to those behind the transformations of Chapter 5.

8.2.1 The parallel program and its simulated-parallel version

The target parallel program. The goal of the transformation process is a parallel program with the following characteristics:

- The program is a collection of N sequential, deterministic processes.
- Processes do not share variables; each has a distinct address space.
- Processes interact only through sends and blocking receives on single-reader–single-writer channels with infinite slack (i.e., infinite capacity).
- An execution is a fair interleaving of actions from processes.

Observe that such a program is consistent with the model of parallel composition with message-passing presented in Chapter 5.

The simulated-parallel version. We can simulate execution of such a parallel program as follows:

- Simulate concurrent execution by interleaving actions from (simulated) processes.
- Simulate separate address spaces by defining a set of distinct address-space data structures.
- Simulate communication over channels by representing channels as queues, taking care that no attempt is made to read from a channel unless it is known not to be empty.

Figure 8.1 illustrates the relationship between the simulated-parallel and parallel versions of a program.

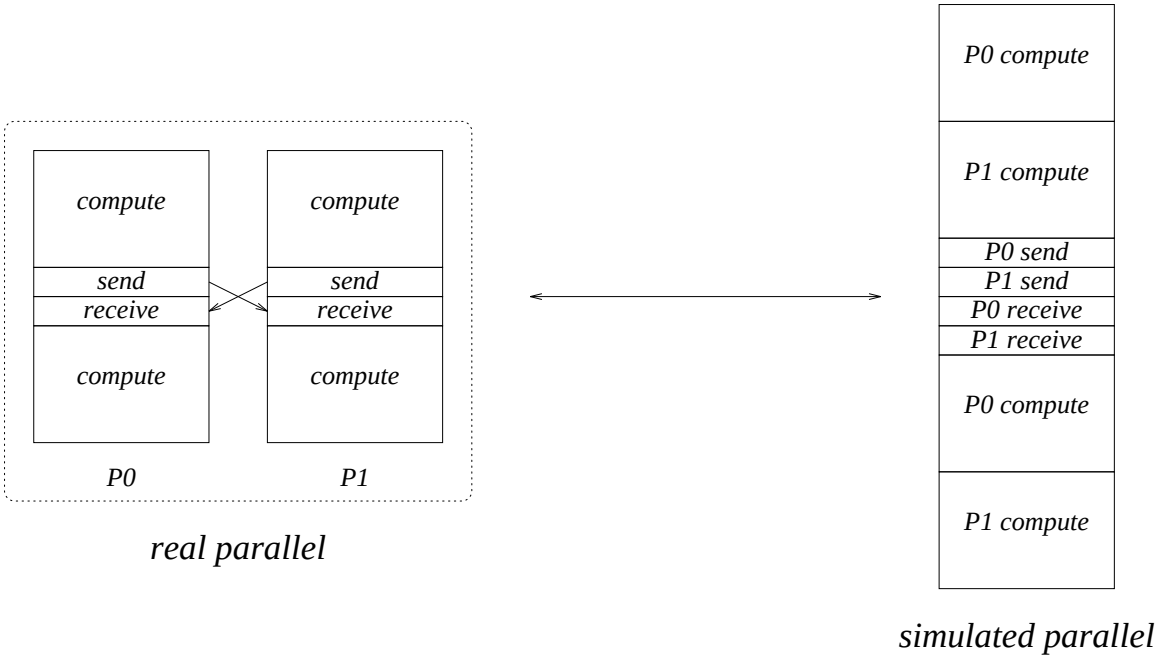


Figure 8.1: Correspondence between parallel and simulated-parallel program versions.

8.2.2 The theorem

Theorem 8.1.

Given deterministic processes $P_0, \ldots, P_{N-1}$ with no shared variables except single-reader–single-writer channels with infinite slack, if I and I' are two maximal interleavings of the actions of the P_j's that begin in the same initial state, I and I' terminate in the same final state.

□

Proof of Theorem 8.1.

Given interleavings I and I' beginning in the same state, we show that I' can be permuted to match I without changing its final state. The proof is by induction on the length of I.

Notation: We write b_i to denote the i-th action of interleaving I and $a_{j,k}$ to denote the k-th action taken by process P_j.

Base case: The length of I is 1. Trivial.

Inductive step: Suppose we can permute I' such that the first n steps of the permuted I' match the first n steps of I. Then we must show that we can further permute I' so that the first $n+1$ steps match I. That is, suppose we have the following situation:

$$\begin{aligned} I: &\quad b_0, b_1, \ldots, b_{n-1}, \quad a_{j,k}, \ldots \\ \textit{permuted } I': &\quad b_0, b_1, \ldots, b_{n-1}, \quad a_{j',k'}, \ldots \end{aligned}$$

Then we want to show that we can further permute I' so that its $n+1$-th action is also $a_{j,k}$.

Observe first that the first action taken in P_j after b_{n-1} in the permuted I' must be $a_{j,k}$: All processes are deterministic, the state after n actions is the same in I and the permuted I', and channels are single-reader–single-writer. Observe analogously that $a_{j',k'}$ is the first action taken in $P_{j'}$ after b_{n-1} in I.

Thus, if $j = j'$, $a_{j,k} = a_{j',k'}$, and we are done. So suppose $j \neq j'$.

Lemma: Observe that for any two consecutive actions $a_{m,n}$ and $a_{m',n'}$, if $m \neq m'$ and it is not the case that $a_{m,n}$ and $a_{m',n'}$ are both actions on some channel c, then because the system contains no shared variables except the channels, these actions can be performed in either order with the same results.

We now demonstrate via a case-by-case analysis that we can permute I' by repeatedly exchanging $a_{j,k}$ with its immediate predecessor until it follows b_{n-1} (as it does in I).

- If $a_{j,k}$ is the m-th receive on some channel c and $a_{j',k'}$ also affects c, then:

  $a_{j',k'}$ is the m'-th send on c, with $m' > m$. (The action is a send because channels are single-reader, and $m' > m$ since $a_{j,k}$ precedes $a_{j',k'}$ in I.) Further, no action between $a_{j',k'}$ and $a_{j,k}$ can affect c (since channels are single-reader–single-writer). Thus, using the lemma, we can repeatedly exchange $a_{j,k}$ with its predecessors in permuted I', up to and including $a_{j',k'}$, as desired.

- If $a_{j,k}$ is the m-th send on some channel c and $a_{j',k'}$ also affects c, then:

  $a_{j',k'}$ is the m'-th receive on c, with $m' < m$. (The action is a receive because channels are single-writer, and $m' < m$ since $a_{j',k'}$ precedes $a_{j,k}$ in permuted I'.) Further, no action between $a_{j',k'}$ and $a_{j,k}$ can affect c (since channels are single-reader–single-writer). Thus, we can repeatedly exchange $a_{j,k}$ with its predecessors in permuted I', up to and including $a_{j',k'}$, as desired.

- If $a_{j,k}$ is the m-th receive on some channel c and $a_{j',k'}$ does not affect c, then:

  If no action between $a_{j',k'}$ and $a_{j,k}$ in permuted I' affects c, then we can perform repeated exchanges as before. If some action b_i does affect c, then it must be the m'-th send, with $m' > m$. (The action is a send because channels are single-reader, and $m' > m$ because the placement of $a_{j,k}$ in I guarantees that actions $b_0, \ldots, b_{n-1}$ contain at least m sends on c.) We can thus exchange b_i with $a_{j,k}$, giving the desired result.

- If $a_{j,k}$ is the m-th send on some channel c and $a_{j',k'}$ does not affect c, then:

  If no action between $a_{j',k'}$ and $a_{j,k}$ in permuted I' affects c, then we can perform repeated exchanges as before. If some action b_i does affect c, then it must be a receive (since channels are single-writer), so it also can be exchanged with $a_{j,k}$, giving the desired result.

- If $a_{j,k}$ is not an action on a channel, then from the lemma we can exchange it with its predecessors as desired.

□

8.2.3 Implications and application of the theorem

Theorem 8.1 implies that if we can produce a sequential simulated-parallel program that meets that same specification as a sequential program, then we can mechanically convert it into a parallel program that meets the same specification, by transforming the simulated processes into real processes, the simulated multiple address spaces into real multiple address spaces, and the simulated communication actions into real communication actions[2]. In general, producing such a simulated-parallel program could be tedious, time-consuming, and error-prone. However, if we start with a program that fits an archetype, as discussed in Section 7.1, and produce a sequential simulated-parallel version of the form described in Section 8.1, where the data-exchange operations correspond to the communication operations of the archetype, then the task becomes manageable. Figure 8.2 illustrates the relationship between the simulated-parallel and parallel versions of such a program.

[2] Observe that a sequence of messages over a single-reader–single-writer channel from process P_i to process P_j can be implemented as a sequence of point-to-point messages from P_i to P_j by giving each message a tag indicating its originating process and receiving selectively based on these tags.

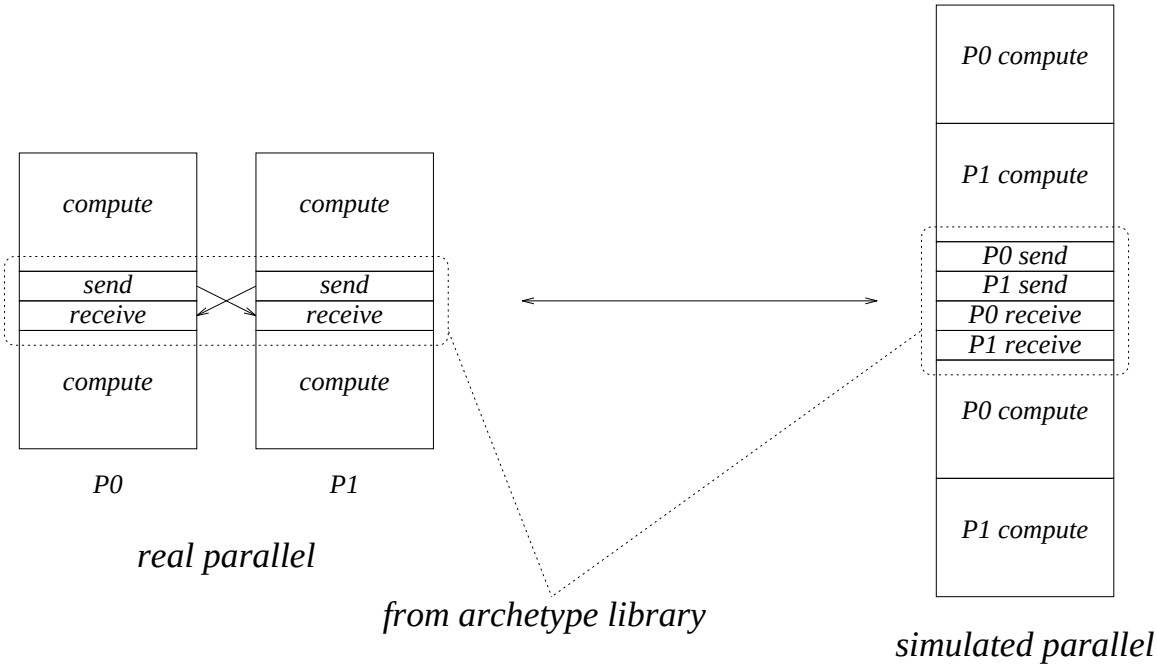


Figure 8.2: Correspondence between parallel and simulated-parallel program versions of archetype-based program.

Each data-exchange operation can be replaced, as described earlier, with a collection of sends and receives: Because of the **arb**-compatibility restrictions, the set of assignments making up a data-exchange operation can be implemented as a set of send–receive pairs over single-reader–single-writer channels, where each assignment generates one send–receive pair, or for efficiency all assignment statements with left-hand-side variables in process P_i's local data and right-hand-side variables in process P_j's local data can be combined into one send–receive pair from process P_j to process P_i. Further, it is straightforward to choose an ordering for the simulated-parallel version that does not violate the restriction that we may not read from an empty channel: First perform all sends; then perform all receives. Finally, if the data-exchange operation corresponds to an archetype communication routine, it can be encapsulated and implemented as part of the archetype library of routines, which can be made available in both parallel and simulated-parallel versions. The application developer thus need not write out and transform the parts of the application that correspond to data-exchange operations.

8.3 Application experiments

As noted earlier, our experiment with this methodology consisted of applying it independently to two sequential implementations of an electromagnetics application. This section describes the application

and the experiment.

8.3.1 The application

The application parallelized in this experiment is an electromagnetics code that uses the finite-difference time-domain (FDTD) technique to model transient electromagnetic scattering and interactions with objects of arbitrary shape and composition. With this technique, the object and surrounding space are represented by a 3-dimensional grid of computational cells. An initial excitation is specified, after which electric and magnetic fields are alternately updated throughout the grid. By applying a near-field to far-field transformation, these fields can also be used to derive far fields, e.g., for radar cross section computations. Thus, the application performs two kinds of calculations:

- **Near-field calculations.** This part of the computation consists of a time-stepped simulation of the electric and magnetic fields over the 3-dimensional grid. At each time step, we first calculate the electric field at each point based on the magnetic fields at the point and neighboring points, and then we similarly calculate the magnetic fields based on the electric fields.
- **Far-field calculations.** This part of the computation uses the above-calculated electric and magnetic fields to compute radiation vector potentials at each time step by integrating over a closed surface near the boundary of the 3-dimensional grid. The electric and magnetic fields at a particular point on the integration surface at a particular time step affect the radiation vector potential at some future time step (depending on the point's position); thus, each calculated vector potential (one per time step) is a double sum, over time steps and over points on the integration surface.

Two versions of this code were available: a public-domain version ("version A", described in [49]) that performs only the near-field calculations, and an export-controlled version ("version C", described in [11]) that performs both near-field and far-field calculations. The two versions were sufficiently different that we parallelized them separately, producing two parallelization experiments.

8.3.2 Parallelization strategy

In most respects, this application fits the pattern of the mesh archetype of Section 7.2.3. Clearly, the near-field calculations are a perfect example of this archetype and thus can be readily parallelized — all that is required is to partition the data and insert calls to nearest-neighbor communication routines.

The far-field calculations fit the archetype less well and are thus more difficult to parallelize. The simplest approach to parallelization involves reordering the sums being computed: Each process

computes local double sums (over all time steps and over points in its subgrid); at the end of the computation, these local sums are combined. The effect is to re-order, but not otherwise change, the summation. This method has the advantages of being simple and readily implemented using the mesh archetype (since it consists mostly of local computation, with one final global-reduction operation). It has the disadvantage of being nondeterministic — that is, not guaranteed to give the same results when executed with different numbers of processes — since floating-point arithmetic is not associative. Nonetheless, because of its simplicity, we chose this method for an initial parallelization.

8.3.3 Applying our methodology

Determining how to apply the strategy. First, we determined how to apply the parallelization strategy, guided by documentation [57] for the mesh archetype, as follows:

- Identify which variables should be distributed (among grid processes) and which duplicated (across all processes). For those variables that are to be distributed, determine which ones should be surrounded by a ghost boundary. Conceptually partition the data to be distributed into "local sections", one for each grid process.

- Identify which parts of the computation should be performed in the host process and which in the grid processes, and also which parts of the grid-process computation should be distributed and which duplicated. Also identify any parts of the computation that should be performed differently in the individual grid processes (e.g., calculations performed on the boundaries of the grid).

Generating the simulated sequential-parallel version. We then applied the following transformations to the original sequential code to obtain a simulated sequential-parallel version, operating separately on the two versions of the application described in Section 8.3.1.

1. In effect partition the data into distinct address spaces by adding an index to each variable. The value of this index constitutes a simulated process ID. At this point all data (even variables that are eventually to be distributed) is duplicated across all processes.

2. Adjust the program to fit the archetype pattern of blocks of local computation alternating with data-exchange operations.

3. Separate each local-computation block into a simulated-host-process block and a simulated-grid-process block.

4. Separate each simulated-grid-process block into the desired N simulated-grid-process blocks. This implies the following changes:

- Modify loop bounds so that each simulated grid process modifies only data corresponding to its local section. This step was complicated by the fact that loop counters in the original code were used both as indices into arrays that were to be distributed and to indicate a grid point's global position, and although the former usage must be changed in this step, the latter must not.
- If there are calculations that must be done differently in different grid processes (e.g., boundary calculations), ensure that each process performs the appropriate calculations.
- Insert data-exchange operations (calls to appropriate archetype library routines).

The result of these transformations was a sequential simulated-parallel version of the original program.

Generating the parallel program. Finally, we transformed this sequential simulated-parallel version into a program for message-passing architectures, as described in Section 8.2.3.

8.3.4 Results

Correctness. For those parts of the computation that fit the mesh archetype — the "near-field" calculations — the sequential simulated-parallel version produced results identical to those of the original sequential code. For those parts of the computation that did not fit well — the "far-field" calculations — the sequential simulated-parallel version produced results markedly different from those of the original sequential code. Our original assumption — that we could regard floating-point addition as associative and thus reorder the required summations without markedly changing their results — proved to be incorrect[3]. Correct parallelization of these calculations would thus require a more sophisticated strategy than that suggested by the mesh archetype, which we did not pursue due to time constraints. While disappointing, this result does not invalidate our hypothesis, since the hypothesis says nothing about using an archetype to parallelize an application that does not fit the archetype pattern well.

For all parts of the computation, however, the message-passing programs produced results identical to those of the corresponding sequential simulated-parallel versions, on their first execution.

Performance. Both versions of the application were parallelized using our Fortran M implementation of the mesh archetype. Because of export-control constraints, we were able to obtain performance data for Version C only on a network of workstations. Table 8.1, Table 8.2, Table 8.3, and Table 8.4 show execution times and speedups for Version C, executing on a network of Sun workstations. Speedup is defined as execution time for the original sequential code divided by execution

[3]Analysis of the values involved showed that they ranged over many orders of magnitude, so it is not surprising that the result of the summation was markedly affected by the order of summation.

time for the parallel code. Figure 8.3 and Figure 8.4 show execution times and speedups for Version A, executing on an IBM SP. The fall-off of performance for more than 4 processors in Figure 8.3 is probably due to the ratio of computation to communication falling below that required to give good performance. Unsurprisingly, performance for the larger problem size (Figure 8.4) scales acceptably for a larger number of processors than performance for the smaller problem size (Figure 8.3), but also falls off when the ratio of computation to communication decreases below that required to give good performance.

	Execution time (seconds)	Speedup
Sequential	78.6	1.00
Parallel, P=1	189.0	0.41
Parallel, P=2	51.4	1.52
Parallel, P=4	25.3	3.10

Table 8.1: Execution times and speedups for electromagnetics code (version C), for 33 by 33 by 33 grid, 128 steps, using Fortran M on a network of Suns.

	Execution time (seconds)	Speedup
Sequential	4309.5	1.00
Parallel, P=4	1189.8	3.62

Table 8.2: Execution times and speedups for electromagnetics code (version C), for 65 by 65 by 65 grid, 1024 steps, using Fortran M on a network of Suns.

	Execution time (seconds)	Speedup
Sequential	123.1	1.00
Parallel, P=1	258.5	0.47
Parallel, P=2	65.4	1.88
Parallel, P=4	32.5	3.78

Table 8.3: Execution times and speedups for electromagnetics code (version C), for 46 by 36 by 36 grid, 128 steps, using Fortran M on a network of Suns.

Ease of use. It is difficult to define objective measures of ease of use, but our experiences in the experiments described in this chapter suggest that the parallelization methodology described herein is not unreasonably difficult to use:

Starting in both cases with unfamiliar code (about 2400 lines for Version C and 1400 lines for Version A, including comments and whitespace), we were able to perform the transformations described

	Execution time (seconds)	Speedup
Sequential	16019.4	1.00
Parallel, P=4	3558.5	4.5

Table 8.4: Execution times and speedups for electromagnetics code (version C), for 91 by 71 by 71 grid, 2048 steps, using Fortran M on a network of Suns.

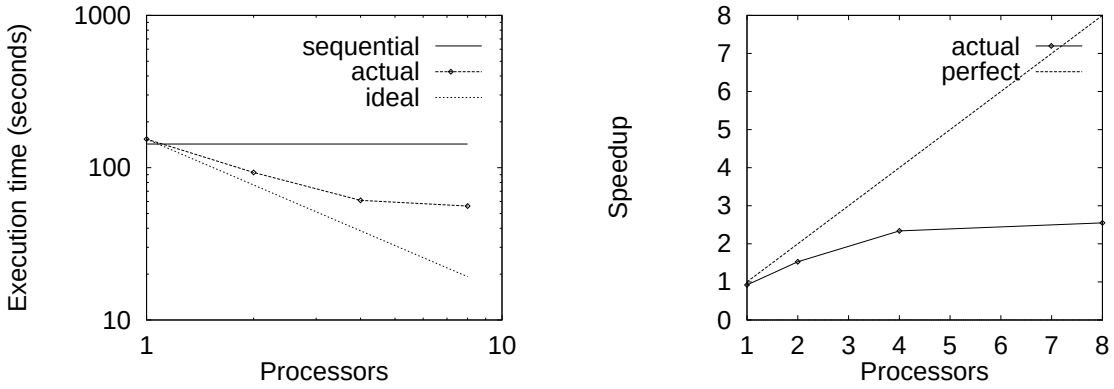


Figure 8.3: Execution times and speedups for electromagnetics code (version A) for 34 by 34 by 34 grid, 256 steps, using Fortran M on the IBM SP.

in Section 8.3.3 relatively quickly: For version C of the code, one person spent 2 days determining how to apply the mesh-archetype parallelization strategy, 8 days converting the sequential code into the sequential simulated-parallel version, and less than a day converting the sequential simulated-parallel version into a message-passing version. For version A of the code, one person spent less than a day determining how to apply the parallelization strategy, 5 days converting the sequential code into the sequential simulated-parallel version, and less than a day converting the sequential simulated-parallel version into a message-passing version.

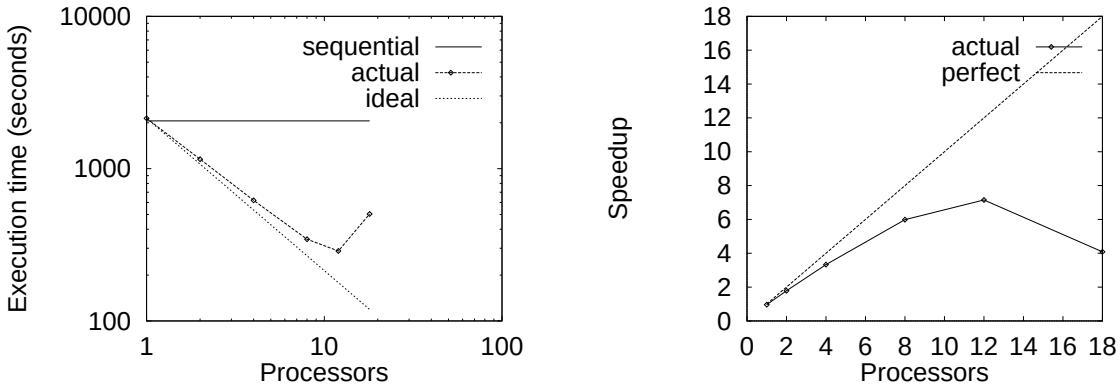


Figure 8.4: Execution times and speedups for electromagnetics code (version A) for 66 by 66 by 66 grid, 512 steps, using Fortran M on the IBM SP.

8.4 Appendix: Details of the conversion process

Figure 8.5 illustrates the overall strategy used to package the conversion from sequential to parallel: Use preprocessor directives to allow generating process code (for both simulated and real parallel versions) and driver code (for the simulated parallel version) from the same source. Figure 8.6, Figure 8.7, and Figure 8.8 show an example.

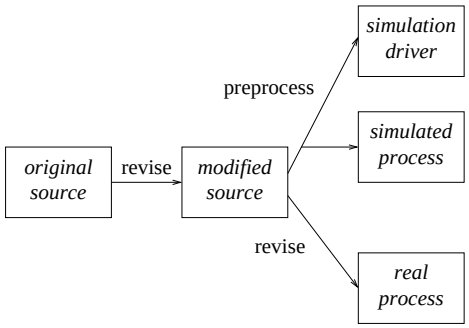


Figure 8.5: Packaging strategy: overview.

```
call local_compute_1(A,B)
call local_compute_2(B,A)
```

Figure 8.6: Packaging strategy: sequential code.

```
call local_compute_1(A,B)
call boundary_exchange(B)
call local_compute_2(B,A)
call boundary_exchange(A)
```

Figure 8.7: Packaging strategy: desired parallel code.

```
#ifdef PROC
    goto (10000, 20000), proc_step
#endif PROC
#ifdef DRIVER
    proc_step = 1
    include 'mesh_doall.h'
#endif DRIVER
#ifdef PROC
10000   continue
    call local_compute_1(A,B)
    return
#endif PROC
#ifdef DRIVER
    call boundary_exchange(B)
#endif DRIVER
#ifdef DRIVER
    proc_step = 2
    include 'mesh_doall.h'
#endif DRIVER
#ifdef PROC
20000   continue
    call local_compute_2(B,A)
    return
#endif PROC
#ifdef DRIVER
    call boundary_exchange(A)
#endif DRIVER
```

Figure 8.8: Packaging strategy: revised source code.

Chapter 9

Related and complementary work

Program specifications and refinement. We take our notion of program specifications from some of the standard ways of giving program specifications for sequential program, for example those of Hoare [44] and Dijkstra [36]. We differ from much work on reasoning about parallel programs, for example Chandy and Misra [24] and Lamport [50], in emphasizing sequential-style specifications over specifications describing ongoing behavior (e.g., safety and progress properties). Our emphasis on program development by stepwise refinement builds on the work of Back [6], Gries [42], and Hoare [44] for sequential programs, and Back [5], Martin [56], and Van de Velde [74] for parallel programs.

Sequential programming models. We base our programming model on the standard sequential model as defined for example by Gries [42].

Parallel programming models. Since we are more interested in sequential-style specifications than in those involving ongoing behavior, our work differs considerably from much other work on parallel programming, for example that of Chandy and Misra (UNITY) [24] and Hoare (CSP) [45]. Other work on parallel programs with sequential equivalents includes that of Valiant (BSP) [73] and Thornley [71]; the former, however, emphasizes performance analysis over analysis of correctness, while the latter focuses mostly on programs for a shared-memory model.

Operational models of program semantics. Our operational model adapts the ideas of state-transitions systems, as described in Chandy and Misra [24], Lynch and Tuttle [52], Lamport [51], Manna and Pnueli [54], and Pnueli [61]; we give a formulation of these ideas that is aimed at facilitating our proofs.

Automatic parallelizing compilers. Much effort has gone into development of compilers that automatically recognize potential concurrency and emit parallel code, for example Fortran D [28] and

HPF [43]. The focus of such work is on the automatic detection of exploitable parallelism, while our work addresses how to exploit parallelism once it is known to exist. Our theoretical framework could be used to prove not only manually-applied transformations but also those applied by parallelizing compilers.

Design patterns. Many researchers have investigated the use of patterns in developing algorithms and applications. Our previous work [20, 23] explores a more general notion of archetypes and their role in developing both sequential and parallel programs. Gamma et al. [41] address primarily the issue of patterns of computation, in the context of object-oriented design. Our notion of a parallel program archetype, in contrast, includes patterns of dataflow and communication. Schmidt [66] focuses more on parallel structure, but in a different context from our work and with less emphasis on code reuse. Shaw [67] examines higher-level patterns in the context of software architectures. Brinch Hansen's work on parallel structures [15] is similar in motivation to our work, but his model programs are typically more narrowly defined than our archetypes. Other work addresses lower-level patterns, as for example the use of templates to develop algorithms for linear algebra in [10].

Program skeletons. Much work has also been done on structuring programs by means of *program skeletons*, including that of Cole [27], Botorog and Kuchen [13, 14], and Darlington et al. [32]. This work is more oriented toward functional programming than ours, although [27] mentions the possibility of expressing the idea of program skeletons in imperative languages, and [14] combines functional skeletons with sequential imperative code.

This work, like that of Brinch Hansen, describes a program development strategy that consists of filling in the "blanks" of a parallel structure with sequential code. Our approach is similar, but we allow the sequential code to reference the containing parallel structure, as in the mesh-spectral archetype examples.

Program development strategies. Fang [39] describes a programming strategy similar to ours, but with less focus on the identification and exploitation of patterns. The Basel approach [16] is more concerned with developing and exploiting a general approach for classifying and dealing with parallel programs. Ballance et al. [9] are more explicitly concerned with the development of tools for application support; while our work can be exploited to create such tools, it is not our primary focus. Kumaran and Quinn [48] focus more on automated conversion of template-based applications into efficient programs for different architectures.

Dataflow patterns. Other work, e.g., Dinucci and Babb [38], has addressed the question of structuring parallel programs in terms of dataflow; our work differs in that it addresses patterns of both dataflow and computation.

Distributed objects. The mesh-spectral archetype is based to some extent on the idea of distributed objects, as discussed for example in work on pC++ [12] and POOMA [4]. Our work on archetypes differs from this work in that we focus more on the pattern of computation and on identifying and exploiting patterns of computation and communication.

Communication libraries. Many researchers have investigated and developed reusable general libraries of communication routines; MPI [58] is a notable example. Others have developed more specialized libraries, for example MPI-RGL [8] for regular grids. We differ from this work, again, in that our focus is on identifying and exploiting patterns.

Chapter 10

Conclusions and directions for future work

10.1 Summary

The specific contribution of the work presented in this thesis is to present a unified theory/practice framework for our approach to parallel program development, tying together the underlying theory, the program transformations, and the program-development methodology. The work described in this thesis falls into two categories, one oriented toward theory and one oriented toward practice.

Theory-oriented work. The theory-oriented work presented in this thesis includes:

- An operational model of programs as state-transition systems that we believe forms a suitable framework for reasoning about program correctness and transformations, particularly transformations between different programming models, as for example between the standard sequential model and the sequential model extended with the unsynchronized parallel composition of Chapter 2. The proofs of Section 2.7 demonstrate that our model can be used as the basis for rigorous and detailed proofs, and while it is beyond the scope of this thesis to include similarly detailed proofs in Chapter 4 and Chapter 5, we believe that the model as presented would be a good framework for such proofs.

- A programming model based on identifying groups of program elements whose sequential composition and parallel composition are semantically equivalent, and a collection of transformations for converting programs in this model to programs for typical parallel architectures. This work provides a framework for program development that permits much of the work to be done with well-understood and familiar sequential tools and techniques.

Practice-oriented work. The practical work presented in this thesis includes:

- Experiments in combining the theory part of the thesis with the notions of archetypes and encapsulation to provide practical assistance to application developers by emphasizing patterns and reuse over full generality. These experiments produced a small number of prototype archetype implementations that proved to be useful in application development. It remains to be seen how broad a range of applications can ultimately be addressed by defining additional archetypes, but these early results are promising.

- Experiments in applying the stepwise-refinement parts of the thesis to practical problems. In the experiments, all transformations were performed manually, which is probably too cumbersome to be an attractive method of parallelizing large programs, but many transformations appear to offer significant potential for automation, and a partially-automated transformation process could be of interest to application developers. It is particularly encouraging that the key transformation from a sequential simulated-parallel program to a real parallel program worked as advertised and appears to be a good candidate for automation.

10.2 Directions for future work

Many promising directions for further work suggest themselves, among them:

- Giving rigorous and detailed proofs of the theorems and transformations of Chapter 4 and Chapter 5. Implicit in the working out of these proofs would be an evaluation of the merits of our overall operational model.

- Applying and extending the model to explicitly address questions of compositionality and modularity. Both are important considerations in practical programming, so again such an investigation would provide feedback on the strengths and limitations of our model.

- Investigating automated support for the transformations presented in this thesis, perhaps by way of a cooperative effort with researchers working on parallelizing compilers.

- Applying the ideas behind the transformations of Chapter 5 — in which barrier synchronization is replaced with the seemingly weaker and yet in this context equally effective pairwise synchronization provided by message-passing — to develop similar transformations from barrier synchronization to weaker synchronization primitives for shared memory (e.g., locks).

- Exploring the range of archetype-guided programming by developing additional archetypes and archetype-based applications.

- Stating and proving additional program transformations, particularly for typical optimizations such as the reuse of storage for different data distributions.

Bibliography

[1] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran 90 Handbook: Complete ANSI/ISO Reference*. Intertext Publications : McGraw-Hill Book Company, 1992.

[2] G. R. Andrews. *Concurrent Programming: Principles and Practice*. The Benjamin/Cummings Publishing Company, Inc., 1991.

[3] ANSI Technical Committee X3H5. X3H5 parallel extensions for Fortran, document number X3H5/93-SD1-Revision M. ⟨ `http://www.kai.com/hints/ftn_m.ps.Z.uu` ⟩, April 1994. Accessed July 1996.

[4] S. Atlas, S. Banerjee, J. C. Cummings, P. J. Hinker, M. Srikant, J. V. W. Reynders, and M. Tholburn. POOMA: A high performance distributed simulation environment for scientific applications. ⟨ `http://www.acl.lanl.gov/PoomaFramework/papers/SCPaper-95.html` ⟩, 1995.

[5] R. J. R. Back. Refinement calculus, part II: Parallel and reactive programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93. Springer-Verlag, 1990.

[6] R. J. R. Back and J. von Wright. Refinement calculus, part I: Sequential nondeterministic programs. In *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*, pages 42–66. Springer-Verlag, 1990.

[7] R. Bagrodia, K. M. Chandy, and M. Dhagat. UC — a set-based language for data-parallel programming. *Journal of Parallel and Distributed Computing*, 28(2):186–201, 1995.

[8] C. F. Baillie, O. Bröker, O. A. McBryan, and J. P. Wilson. MPI-RGL: a regular grid library for MPI. ⟨ `http://www.cs.colorado.edu/~ broker/mpi_rgl/mpi_rgl.ps` ⟩, 1995.

[9] R. A. Ballance, A. J. Giancola, G. F. Luger, and T. J. Ross. A framework-based environment for object-oriented scientific codes. *Scientific Programming*, 2(4):111–121, 1993.

[10] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1993.

[11] J. H. Beggs, R. J. Luebbers, D. Steich, H. S. Langdon, and K. S. Kunz. User's manual for three-dimensional FDTD version C code for scattering from frequency-independent dielectric and magnetic materials. Technical report, The Pennsylvania State University, July 1992.

[12] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3):7–22, 1993.

[13] G. H. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In L. Bouge, editor, *Proceedings of EuroPar '96*, volume 1123–1124 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[14] G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996.

[15] P. Brinch Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency: Practice and Experience*, 5(5):407–423, 1993.

[16] H. Burkhart, R. Frank, and G. Hächler. Structured parallel programming: How informatics can help overcome the software dilemma. *Scientific Programming*, 5(1):33–45, 1996.

[17] R. M. Butler and E. L. Lusk. Monitors, messages, and clusters — the p4 parallel programming system. *Parallel Computing*, 20(4):547–564, 1994.

[18] C. Canuto. *Spectral Methods in Fluid Dynamics*. Springer Series in Computational Physics. Springer-Verlag, 1988.

[19] P. Carlin, K. M. Chandy, and C. Kesselman. The Compositional C++ language definition. Technical Report CS-TR-92-02, California Institute of Technology, 1992.

[20] K. M. Chandy. Concurrent program archetypes. In *Proceedings of the Scalable Parallel Library Conference*, 1994.

[21] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming language. Technical Report CS-TR-92-01, California Institute of Technology, 1992.

[22] K. M. Chandy and L. Lamport. Distributed snapshots — determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3:63–75, 1985.

[23] K. M. Chandy, R. Manohar, B. L. Massingill, and D. I. Meiron. Integrating task and data parallelism with the group communication archetype. In *Proceedings of the 9th International Parallel Processing Symposium*, 1995.

[24] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1989.

[25] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39:472–482, 1936.

[26] M. J. Clement and M. J. Quinn. Overlapping computations, communications and I/O in parallel sorting. *Journal of Parallel and Distributed Computing*, 28(2):162–172, August 1995.

[27] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[28] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The Parascope parallel programming environment. *Proceedings of the IEEE*, 82(2):244–263, 1993.

[29] D. Dabdub and R. Manohar. Performance and portability of an air quality model. *Parallel Computing*, 1997. To appear in special issue on regional weather models.

[30] D. Dabdub and J. H. Seinfeld. Air quality modeling on massively parallel computers. *Atmospheric Environment*, 28(9):1679–1687, 1994.

[31] D. Dabdub and J. H. Seinfeld. Parallel computation in atmospheric chemical modeling. *Parallel Computing*, 22:111–130, 1996.

[32] J. Darlington, A. J. Field, P. O. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. White. Parallel programming using skeleton functions. In A. Bode, editor, *Proceedings of PARLE 1993*, volume 694 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[33] G. Davis. Spectral methods for scientific computing. ⟨ `http://www.etext.caltech.edu/Implementations/Spectral/spectral.html` ⟩, 1994.

[34] G. Davis and B. Massingill. The mesh-spectral archetype. Technical Report CS-TR-96-26, California Institute of Technology, 1997. Also available via ⟨ `http://www.etext.caltech.edu/Implementations/` ⟩.

[35] E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivations of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[36] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Inc., 1976.

[37] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Springer-Verlag, 1990.

[38] D. C. Dinucci and R. G. Babb II. Development of portable parallel programs with Large-Grain Data Flow 2. In G. Goos and J. Hartmanis, editors, *CONPAR 90 — VAPP IV*, volume 457 of *Lecture Notes in Computer Science*, pages 253–264. Springer-Verlag, 1990.

[39] N. Fang. Engineering parallel algorithms. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, 1996.

[40] I. T. Foster and K. M. Chandy. FORTRAN M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):24–35, 1995.

[41] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[42] D. Gries. *The Science of Programming*. Springer-Verlag, 1981.

[43] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. *Scientific Programming*, 2(1–2):1–170, 1993.

[44] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[45] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[46] International Standards Organization. ISO/IEC 1539:1991 (E), Fortran 90, 1991.

[47] E. Isaacson and H. B. Keller. *Analysis of Numerical Methods*. John Wiley & Sons, Inc., 1966.

[48] S. Kumaran and M. J. Quinn. An architecture-adaptable problem solving environment for scientific computing, 1996. Submitted to *Journal of Parallel and Distributed Computing*.

[49] K. S. Kunz and R. J. Luebbers. *The Finite Difference Time Domain Method for Electromagnetics*. CRC Press, 1993.

[50] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 2:125–143, 1977.

[51] L. Lamport. A temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[52] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, 1987.

[53] B. J. MacLennan. *Functional Programming: Practice and Theory.* Addison-Wesley, 1990.

[54] Z. Manna and A. Pnueli. Completing the temporal picture. *Theoretical Computer Science*, 83(1):97–130, 1991.

[55] A. J. Martin. An axiomatic definition of synchronization primitives. *Acta Informatica*, 16:219–235, 1981.

[56] A. J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[57] B. Massingill. The mesh archetype. Technical Report CS-TR-96-25, California Institute of Technology, 1997. Also available via ⟨ `http://www.etext.caltech.edu/Implementations/` ⟩.

[58] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3–4), 1994.

[59] J. M. Morris. Piecewise data refinement. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs.* Addison-Wesley Publishing Company, Inc., 1990.

[60] P. Pierce. The NX message-passing interface. *Parallel Computing*, 20(4):463–480, 1994.

[61] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[62] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1988.

[63] D. I. Pullin. Direct simulation methods for compressible ideal gas flow. *Journal of Computational Physics*, 34:231, 1980.

[64] A. Rifkin and B. L. Massingill. Performance analysis for mesh and mesh-spectral archetype applications. Technical Report CS-TR-96-27, California Institute of Technology, 1997.

[65] R. Samtaney and D. I. Meiron. Hypervelocity Richtmyer-Meshkov instability. *Physics of Fluids*, 9(6):1783–1803, 1997.

[66] D. C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, 1995.

[67] M. Shaw. Patterns for software architectures. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 453–462. Addison-Wesley, 1995.

[68] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, April 1992.

[69] P. A. G. Sivilotti. A verified integration of imperative parallel programming paradigms in an object-oriented language. Technical Report CS-TR-93-21, California Institute of Technology, 1993.

[70] P. A. G. Sivilotti. Reliable synchronization primitives for Java threads. Technical Report CS-TR-96-11, California Institute of Technology, 1996.

[71] J. Thornley. A parallel programming model with sequential semantics. Technical Report CS-TR-96-12, California Institute of Technology, 1996.

[72] J. Thornley and K. M. Chandy. Barriers: Specification. Unpublished document.

[73] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[74] E. F. Van de Velde. *Concurrent Scientific Computing.* Springer-Verlag, 1994.